AFRL-IF-RS-TR-2006-204
Final Technical Report
June 2006

AWDRAT: ARCHITECTURAL DIFFERENCING, WRAPPERS, DIAGNOSIS, RECOVERY, ADAPTIVITY AND TRUST MANAGEMENT

Massachusetts Institute of Technology, CSAIL

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S473

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 074-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503			
1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** JUNE 2006	**3. REPORT TYPE AND DATES COVERED** Final Jun 2004 – Dec 2005	
4. TITLE AND SUBTITLE AWDRAT: ARCHITECTURAL DIFFERENCING, WRAPPERS, DIAGNOSIS, RECOVERY, ADAPTIVITY AND TRUST MANAGEMENT			**5. FUNDING NUMBERS** C - FA8750-04-2-0240 PE - 62304E PR - S473 TA - SR WU - SP
6. AUTHOR(S) Robert Balzer, Howard Shrobe, Neil Goldman, David Wile			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) MIT CSAIL, 77 Massachusetts Ave., 32-225, Cambridge Massachusetts 02138 Teknowledge, Inc., 4640 Admirality Way, Suite 1010, Marina del Rey California 90292			**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Drive, Arlington Virginia 22203-1714 AFRL/IFGA, 525 Brooks Road, Rome New York 13441-4505			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2006-204
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Alan J. Akins/IFGA Alan.Akins@rl.af.mil			
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. PA#06-412			**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

This document is the final report for AWDRAT, an effort in the DARPA funded Self-Regenerative System (SRS) program conducted by MIT and Teknolwedge. AWDRAT stands for Architectural Differencing, Wrappers, Diagnosis, Recovery, Adaptivity, and Trust Management. AWDRAT is a framework that provides survivability services to legacy (or new) applications. It does so by modeling the intended behavior of the application, using wrappers to instrument the application system and using the information derived from the wrappers to detect deviations from the expected behavior. When the application fails to behave as expected, AWDRAT invokes diagnostic services to determine what resources might have been compromised and then updates its trust model to reflect the probabilities of compromised resources. Recovery efforts are guided by the trust model, steering the system away from possibly compromised resources. AWDRAT was shown in both Red-Team and internal experiments to detect and correct failures at a level exceeding the goals of the SRS program.

14. SUBJECT TERMS Regenerative systems, cyber defense, cognitive immunity, diagnostic services, wrappers, adaptive software, trust model			**15. NUMBER OF PAGES** 78
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

List of Figures

List of Tables

1. Technical Approach

1.1 The Challenge

Within MIT CSAIL an ensemble of computers runs a Visual Surveillance and Monitoring application. On January 12, several of the machines experience unusual traffic from outside the lab. Intrusion Detection systems report that several password scans were observed. Fortunately, after about 3 days of varying levels of such activity, things seem to return to normal; for another 3 weeks no unusual activity is noticed. Then, however, one of the machines (named Harding), which is crucial to the application, reports that it is experiencing unusually high load averages and that its application-level software is receiving less than the expected quality of service. The load average, degradation of service, the consumption of disk space and the amount of traffic to and from unknown outside machines continue to increase to annoying levels. Then they level off. On March 2, a second machine in the ensemble (Grant) crashes; fortunately, the application has been written in a way which allows it to adapt to unusual circumstances. The system considers whether it should migrate the computations which would normally have run on Grant to Harding; however, these computations are critical to the application. The system decides that in spite of the odd circumstances noticed on Harding earlier, it is a reasonable choice.

Did the system make a good choice? It turns out it did. The system needed to run those computations somewhere; even though Harding was loaded more heavily than expected, it still represented the best pool of available computational resources. Other machines were even more heavily loaded with other critical computations of the application. But what about all the unusual activity that had been noticed on Harding? It turns out that what had, in fact, transpired is that hackers had gained access to Harding by correctly guessing a password; using this they had set up a public FTP site containing among other things pirated software and erotic imagery. They had not, in fact, gained root access. There was, therefore, no serious worry that the critical computations migrated to Harding would experience any further compromise. (Note: the adaptive system in this story is fictional, the compromised computers reflect an amalgam of several real incidents).

Let's suppose instead that (1) the application was being run to protect a US embassy in Africa during a period of international tension (2) that we had observed a variety of information attacks being aimed at Harding earlier on (3) that at least some of these attacks are of a type known to be occasionally effective in gaining root access to a machine like Harding and that (4) they are followed by a period of no anomalous behavior other than a periodic low volume communication with an unknown outside host. When Grant crashes, should Harding be used as the backup? In this case, the answer might well be the opposite; for it is quite possible that an intruder has gained root access to Harding; it is also possible that the intent of the intrusion is malicious and political. It is less likely, but still possible, that the periodic communication with the unknown outside host is an attempt to contact an outside control source for a “go signal” that will initiate serious spoofing of the application. Under these circumstances, it is wiser to shift the computations to a different machine in the ensemble even though it is considerably more overloaded than Harding.

1.1.1 Architectural Lessons

What can we learn from these examples?

- It is crucial to have a *trust-model* that estimates (1) To what degree and for what purposes a computer (or other computational resource) may be relied on, as this influences decisions about what tasks should be assigned to them; (2) What contingencies should be provided for; and (3) How much effort to spend watching over them.

- Making this estimate depends in turn on having a model of: (1) The possible ways in which a computational resource may be *compromised*; (2) The vulnerabilities possessed by the resources; (3) The general forms of the attacks capable of exploiting these vulnerabilities.

- This in turn depends on having in place a system for long term *monitorin*g *an*d *analysi*s of the computational infrastructure which can detect patterns of activity such as “a period of attacks followed by quiescence followed by increasing degradation of service”. Such a system must be capable of assimilating information from a variety of sources including both self-checking observation points within the application itself and external intrusion detection systems.

- The system and its applications must be capable of *self-monitorin*g *an*d *diagnosi*s and capable of *adaptatio*n so that they can best achieve their purposes using the available resources, even when these resources might be partially compromised.

- This, in turn, depends on the ability of the application, monitoring, and control systems to engage in *cognitiv*e *decisio*n *makin*g about what resources they should use in order to achieve the best balance of expected benefit to risk.

In this project, we have developed a software infrastructure providing services like those just enumerated. Existing application software may be retrofitted to this infrastructure and new applications may be developed against it. This infrastructure, called AWDRAT (for Architectural-differencing, Wrappers, Diagnosis, Recovery, Adaptivity and Trust-modeling), provides a variety of services that are normally taken care of in an *a*d *ho*c manner in each individual application, if at all. These services include fault containment, execution monitoring, diagnosis, recovery from failure and adaption to variations in the trustworthiness of the available resources.

Software systems tethered within the AWDRAT environment behave reflectively and adaptively, and with the aid of the AWDRAT infrastructure these systems regenerate themselves when attacks or mistakes cause serious damage. AWDRAT provides a convenient framework for structuring new application systems, for restructuring legacy code, and for modeling software behavior in such a way that robustness is a natural byproduct, whether the cause of compromise is accidental misconfiguration or intentional coordinated attack.

1.2 Overview of the AWDRAT System Architecture

The discussion above illustrates the need for a software system to recognize and respond adaptively to signs of compromise. Rather than leaving it to the developers of each and every application system to build the complex facilities needed to achieve robustness in the face of attacks or mistakes, we instead have built the AWDRAT infrastructure that provides application level software with the services necessary to achieve robustness. The name AWDRAT enumerates these services: Architectural-differencing, Wrapper generation and placement, Diagnosis, Recovery, Adaptivity and Trust Management. We will present an overview of these services in the rest of this section and then turn in the following sections to a more complete description of each of the facilities.

The architecture of AWDRAT is cognitive, goal driven, self-reflective and adaptive. AWDRAT provides services that allow application software systems to respond in reasonable ways to compromises of the resources, avoiding them if the compromise would cause serious harm, and using compromised resources if they can be employed usefully in pursuit of important goals without fear of damaging properties of interest. Finally, AWDRAT regenerates its hosted applications by removing the compromises (e.g. changing the stolen user password in our scenario), restoring corrupted data sets, instituting defenses against observed attacks, increasing the degree of variability in the hosted software so as to confuse future attackers or when none of this is possible by helping the application software to avoid the compromised resource.

These services do not come for free. Software hosted within the AWDRAT environment must be (re)structured as discrete methods capable of rendering specific services; wherever possible, application software is encouraged to provide multiple methods for the same generic service (as is often the case when systematic domain engineering is undertaken). In addition, semantic meta-data must be provided to AWDRAT for each method. The AWDRAT framework is intended to host distributed application systems and wherever possible the methods should be (re)structured as mobile code, allowing AWDRAT to decide which host is best suited to running a computation.

In the discussion to follow, we use the term “resource” to mean any host, code segment, data set, etc. that is necessary to support a particular computational step; when there are multiple resources that are more or less equivalent, AWDRAT has the freedom to choose between them. The AWDRAT decision cycle is:

- When presented with a task to be achieved, AWDRAT consults its method library, finding all applicable methods relevant to the service request. Meta-data associated with each method describes the method’s resource requirements as well as the quality of service that the method will render given a particular resource selection.

- Each combination of method and supporting resources is evaluated, taking into account the cost of the resources and the value of the service quality delivered. During this evaluation, the trust model is consulted to assess the possibility that the resources are compromised; the cost of a potential failure is then taken into account in assessing the overall expected value of this particular combination of method and resources. The method and set of resources that promise the best overall tradeoff is selected for execution.

- Accompanying each method is an architectural model of the computation performed by the method. This specifies prerequisite and post conditions for each step of the computation as well as state transitions that occur and invariants that are maintained during the step's execution. Wrappers are synthesized to check that these constraints are met during the method's execution. AWDRAT uses these wrappers to effect a technique called *Architectura*l *Differencing*. It interprets the architectural model of the method in parallel with the executing code, noting when the two produce discrepant results or when the executing code violates a constraint of the architectural model. Other wrappers are synthesized to provision back up copies of significant data that will be used in recovery and regeneration.

- If any constraint imposed by the architectural model is violated, model-based diagnosis is invoked to assess what part of the computation may have failed and to assess what resources might have been compromised in such a way as to lead the observed misbehavior. The trust model is updated with the information produced by the diagnosis, leading to new assessments of the reliability and trustability of the computational resources.

- Recoverable data (e.g. databases, code segments, password files) are restored in order to establish a consistent point from which to resume the computation. If specific vulnerabilities are implicated in the failure, then attempts are made to repair the vulnerability.

- AWDRAT also constantly monitors its sensors for evidence that an attack is underway. Attacks are modeled as multi-stage plans and plan recognition techniques are used to assess the probability that an attack has succeeded or is underway. This too updates the trust model.

- AWDRAT returns to the beginning of its decision cycle, trying again to meet the application's goal; however it is now informed by the results of diagnosis. It restarts the computation from any place that the diagnosis guarantees to have been successfully completed and chooses a method and resources in light of the updated trust model.

- During the execution of a method AWDRAT collects data about the method's performance and resource consumption. These data are used to form and refine constraints on the "nonfunctional properties" of the method which will then be monitored during future executions. Deviations from expected performance and resource consumption constraints generate calls for diagnosis and recovery just as would a violation of a hard constraint on the input-output relationships of the method.

This approach guarantees that AWDRAT will find some way to achieve the application's goals if there is an available method; it also guarantees that it will steer the application clear of resources that it has reason to believe are corrupted if the compromise to the resources is likely to cause damage. The application system behaves adaptively when hosted within the AWDRAT environment.

We have called this approach to survivability "Automatic Trust Management" [36, 38, 37]. Prior to the SRS program, we had been investigating its principles and experimenting with prototype implementations of key components. This approach is a cogent example of "cognitive immunity" in that it relies on explicit, symbolic representation and reasoning, an explicit model of the intended functioning of the applications, as well as on a deliberative and reflective self-adaptive computing architecture. AWDRAT unites the following components:

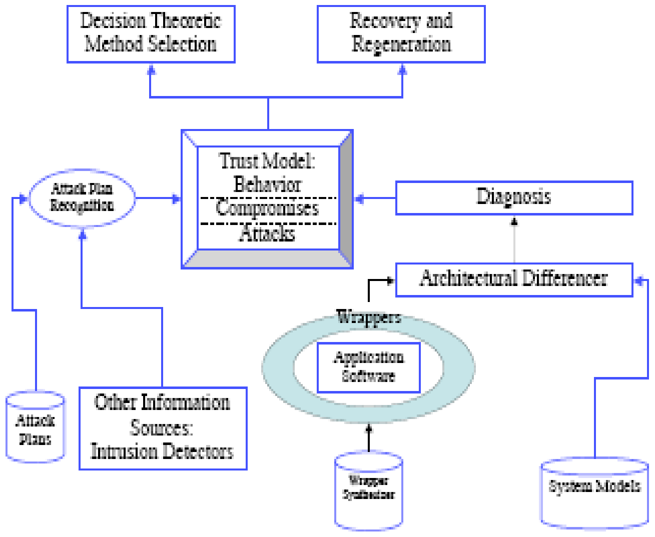


Figure 1: The AWDRAT Architecture

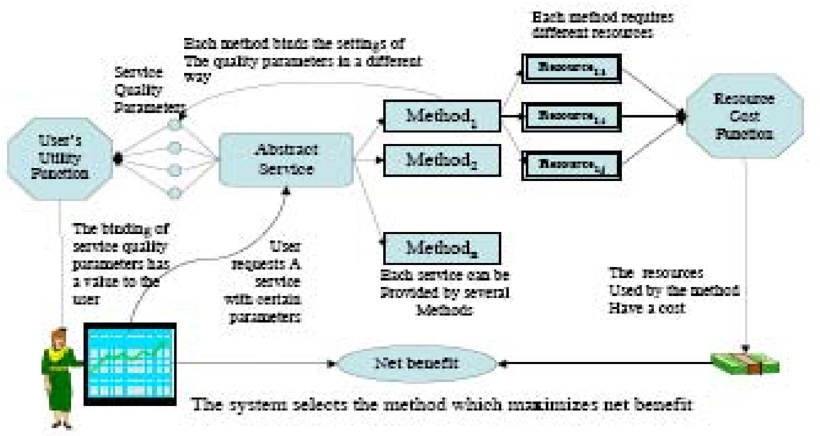

Each method requires different resources
Each method binds the settings of The quality parameters in a different way
Service Quality Parameters
User's Utility Function
Abstract Service
Method1
Method2
Methodn
Resource1,1
Resource1,2
Resource1,3
Resource Cost Function
The binding of service quality parameters has a value to the user
User requests A service with certain parameters
Each service can be Provided by several Methods
The resources Used by the method Have a cost
Net benefit
The system selects the method which maximizes net benefit

Figure 2: Method Selection

- A trust model that assesses the likelihood that each computational resource is compromised in a specific way and thereby implies for what purposes the component may be used reliably. The first prototype for this was developed in the Oasis program [37].

- An infrastructure to support self-adaptive application systems. Such systems will have more than one way to achieve each major sub-task and will dynamically decide how best to achieve each goal in light of current conditions, in particular, in light of the trust model. This draws on work done in both the Oasis program and in MIT's Project Oxygen [36].

- A system modeling framework that allows AWDRAT to operationalize the specifications of an application in terms of conditions expected to hold at particular points and invariants that must be true across intervals of the computation.

- A synthesis system that automatically generates wrappers around components of the computation and around interfaces to the operating system such that the important state of the computation is observable and such that redundant copies of critical data can be automatically provisioned. This is based on both an existing Teknowledge technology drawn from earlier programs [12, 2, 3] and on new technology developed during SRS.

- A diagnostic component that is activated by the detection of a symptom (i.e. the failure of a computational component to behave in accordance with its specification) by some wrapper. The diagnostic component then determines what failures may have led to the observed symptom, whether this failure is indicative of a compromised resource, what the cause of this compromise is likely to have been and whether this cause is likely to have affected other resources as well. In particular, we are concerned with intentional attacks that exploit vulnerabilities of the computational resources[1]. The diagnostic component then updates the trust model to reflect its conclusions. This is based on work begun in the Oasis program [38, 36] and further enhanced during SRS.

- A recovery component, developed during SRS, that operates after the diagnostic component has assessed the cause of the failure. The recovery component is responsible for restoring corrupted data sets to a usably consistent preserved state (often capitalizing on checkpointed data automatically provisioned by wrappers) and for the selecting of a suitable method for achieving the application's goals, given the updated beliefs in the trust model.

1.2.1 The Trust Model

The core representation enabling cognitive immunity is the Trust Model whose role is to inform AWDRAT about which resources may be trusted and for what purposes. It does this by creating a set of models of the ways in which each resource may be compromised and then associating a probability with the normal state as well as each compromised state of each resource (by resource we mean any object, such as a host computer, a data set, a code segment, that is necessary to support the execution of a computation).

We model each resource at many levels of detail, decomposing until we reach a level at which it is convenient to observe evidence of compromises, to describe the types of compromises and to characterize the recovery methods relevant to each compromise. We do this by grounding the analysis in a comprehensive ontology that covers:

- System properties
- System Types
- System structure
- The control and dependency relationships between system components.

The ontology covers what types of computing resources are present in the environment, how the resources are composed from components (e.g. an operating system has a scheduler, a file system, etc.), how the components control one another's behavior, and what vulnerabilities are known to be present in different classes of these components. The analysis begins by asking what are the desirable properties that systems are expected to deliver and how these properties depend on the correct functioning of specific components of the system (for example, predictable performance of a computer system depends on the correct functioning of its scheduler). Typical properties include:

[1] However, this approach is equally valid in reasoning about unintentional but systematic environmental factors that might, for example, degrade physical resources or interfere with communications

- Reliable Performance
- Privacy of Communications
- Integrity of Communications
- Integrity of Stored Data
- Privacy of Stored Data

A relatively simple reasoning process (encoded in a rule-based system) then explores how a desirable property of a system can be impacted (e.g. you can impact the predictability of performance by affecting the scheduler, which in turn can be done by changing its input parameters which in turn can be done by gaining root access which finally is enabled by a buffer overflow attack on a process running with root privileges). The output of this reasoning is a set of multi-stage attack plans, each of which is capable of affecting the property of interest, see section 1.2.4.

We also provide a structural model of the entire computing environment under consideration, including:

- Network structure and topology
  - How is the network decomposed into subnets
  - Which nodes are on which subnets
  - Which routers and switches connect the subnets
  - What types of filters and firewalls provide control of the information flow between subnets
- System types:
  - What type of hardware is in each node
  - How is the hardware decomposed into sub-systems
  - What type of operating system is in each node
  - How is the operating system decomposed into sub-systems
- Server and user software suites: What software functionality is deployed on each node.
- What are the access rights to data and how are they controlled

- What are the places in which data or code is stored or transmitted
- How is data or code transformed from one representation to another

For example, we decompose a computer system into the physical hardware and the running operating system; the operating system is in turn decomposed into its various subsystems such as the file system, the scheduler, log-in manager, etc. We might then identify one type of possible compromise to the scheduler as the imposition of an unfair scheduling policy favoring certain tasks over others. Alternatively we might go one layer deeper, decompose the scheduler into a process priority table and a scheduling algorithm; here the compromise might be in the form of a change in the priority table, favoring a specific process. In general, the strategy is to decompose the model to a level that gives good diagnostic resolution, leads to accurate decisions about how to recover and that requires no more accurate information than we are capable of observing or of inferring from our observations.

The trust model constitutes the middle tier of a three-tiered Bayesian inference system. The highest tier is concerned with the observations and inferences we make about executing computations, the lowest tier is concerned with the observations and inferences we can make about attacks. Intuitively, a compromise to a computational resource will eventually manifest itself in a misbehavior of some computation that relies on that resources (e.g. if the scheduler is compromised then some processes are likely to perform worse than expected; if the access control mechanisms are compromised then some protected data will eventually fail to meet its integrity constraints). An observation of such a misbehavior provides confirmation that the resource is compromised. This reasoning is made precise by translating it into the machinery of Bayesian networks; we create a link (a conditional probability) from a compromised state of a resource to a misbehaving state of the computation (see section on diagnosis on page 17). An observation of such a misbehaving computation will then cause the Bayesian network to increase the probability that the resource is compromised.

Similarly, if we observe evidence of an attack we can infer that it is likely that the attack took advantage of the vulnerabilities of its target resources, leaving these in a compromised state. Within the Bayesian network this is implemented by connecting the node representing such a successful attack with other nodes representing the resulting compromised states of the vulnerable resources. The Bayesian network machinery will then propagate the probability forward from the attack node to the nodes representing the compromised resource state.

In summary, the trust model has three levels: (1) Computational behavior, (2) Resource health status, (3) Attacks and vulnerabilities[2] . It unites these into a single Bayesian inference mechanism that assimilates all and propagates all evidence bidirectionally, connecting the observations of misbehaving computations, to the conclusion that certain resources are likely to have been compromised and connecting these conclusions to the

[2] We note again that this approach is equally applicable to other events such as incorrect settings of configuration files by the system's operators.

further conclusion that certain types of attacks (or mistakes) are likely to have occurred. It also makes bottom-up inferences, starting with evidence of an attack, leading to the inference that certain resources are likely to have been compromised and finally predicting that certain computations are likely to misbehave.

The net result of this Bayesian inference is an updated estimate of the health status of each resource, an indication of what attacks are likely to have compromised some resources (and therefore which are likely to cause harm to other resources in the future) and an explanation for the observed misbehavior of executing computations. These assessments of the health status of computational resource then informs AWDRAT as it makes choices about how to achieve application level goals and about what resources to use in doing so. This is discussed in the section 1.2.2.

1.2.2 Self-Adaptive Software and Rational Decision Making

The purpose of the trust model is to inform AWDRAT in its choice of how to conduct application level computations and with what resources. To capitalize on the trust model, AWDRAT must be structured in a way that facilitates such a choice, making the applications that run within the AWDRAT framework adaptive to changes in the trustworthiness of their resources.

Our approach to adaptation is an explicit, cognitive approach, based in decision theory. We structure the application systems around the services they are capable of rendering (i.e. the goals they can achieve on behalf of their client[3]) and the various methods they have available for achieving those goals (i.e. the plans available for achieving each goal). Each method that is capable of rendering a specific service requires a set of resources satisfying certain constraints (e.g. it needs a set of host computers with adequate memory, networks with appropriate bandwidth, data sets, etc.). These resources may be more or less available at the time of the request; this availability can be reflected as a price.

In addition, each service can be rendered with varying qualities of service; for example, if the service is to display some information, then different methods will vary in terms in the speed with which the information is presented (a printer being much slower than a monitor, for example), the quality of the output (the printer might be much better than a monitor), the privacy of the presentation (a PDA screen would be more private than a projector), etc.

As the context varies, different qualities of service may have more of less value to the requester of the service (at some point privacy might outweigh the quality of the display; at other times, the exact opposite might be true). In other words, the client has preferences over the qualities of service; the degree to which these preferences have been satisfied may be measured using techniques we have developed for converting preference sets into numerical utility functions [28]. In addition, we may assign to each set of resources a price reflecting their current availability. The difference between these two

[3] By clients we mean either an actual end-user or another method that makes a service request

values is called *net-benefit*. AWDRAT will select that method and set of resources that maximizes net-benefit so as to provide adaptivity. We note in passing that this approach is a generalization of the method dispatch of object-oriented programming; instead of selecting methods based on type signatures, we select methods based on net-benefit.

1.2.3 Adaptation to the Trust Model

So far, we have been describing an AWDRAT as an adaptive framework that responds to variability in the availability of resources and the variability of the client's preferences. However, in the current context, we are also interested in adaptation to compromises in the system's resource pool. To do this, we turn to considering the *expecte*d net-benefit of a particular method-resource selection.

The trust model gives us an assessment of the probability that a resource is compromised. In addition, it connects this estimate to assessments of the likelihood that a component of a computation will behave in other than the expected manner as we saw in section 1.2.1. This, in turn, may lead to a different quality of service being delivered by the application, which will affect the estimated benefit of the method. So in thinking about the benefit that

$$EB(M,\vec{R}) = \sum_{\vec{RS_k} \in resource-states(\vec{R})} P(\vec{RS_k}) \bullet U(\vec{SQ}(\vec{RS_k}))$$

$$EBsuccess(M,\vec{R_i}) \equiv P(\wedge PRE_j(M)) \bullet EB(M,\vec{R_i})$$

$$ECFail(M) \equiv (1 - P(\wedge PRE_j(M)) \bullet FailCost(M)$$

$$NetEB(M,\vec{R_i}) = EBsuccess(M,\vec{R_i}) - ECFail(M) - RC(\vec{R_i})$$

$$Selected(M,\vec{R_i}) = \arg\max_{M,\vec{R_i}}(NetEB(M,\vec{R_i}))$$

Where M is a method, R is a set of resources, RS is a set of state assignments to each resource in a set. SQ is the service quality delivered, U is the utility function. EB is Expected Benefit and EC is Expected Cost

Figure 3: Equations governing Method Selection

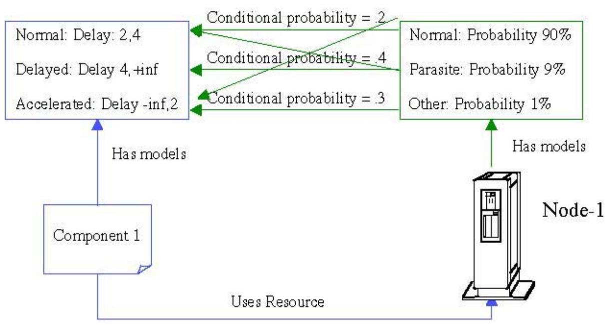


Figure 4: Dependence Between Resource States and Behavioral Mode of Computation

a particular method and set of resources will deliver we must consider all the possible modes (both healthy and compromised) of all the resources used, and for each mode weigh the benefit delivered by the probability that the resources are in this mode. The sum over all combinations is called the *expected net benefit* and is a measure of how well this method and set of resources is likely to meet the client's needs, given the uncertainty we have about the health of the resources. We note that a set of compromised resources can deliver "negative benefit," for example by publicly exposing information intended to be kept private; if the resources are extremely likely to be in such a compromised state, then the expected net benefit will be negative.

We also note that each method has certain prerequisite conditions and that absent these conditions, the method will fail. In addition to assessing the health status of all resources, the trust model also assesses the likelihood that these prerequisite conditions hold (see section 1.2.7). If a computational step is attempted when its prerequisite conditions do not obtain, then the computation will fail and there will be a cost associated with the damage such a failure engenders. Since there is a probability that the prerequisite conditions do hold, we subtract the cost of failure weighted by its probability from the expected net benefit weighted by its probability, yielding a measure of *total net expected benefit*. AWDRAT will choose that method which maximizes total net expected benefit so as to provide adaptivity to resource compromises.

Finally, we must consider the total net expected benefit of not rendering the service at all. This has a cost to the client; but it is possible that the expected net benefit of the best possible method is itself negative (i.e. it's a cost) and worse than cost of doing nothing. This would be the case if it is extremely likely that the resources are corrupted or that the method's prerequisites don't hold and that it will, therefore, fail catastrophically. For example, if you are about to send a message to your neighbor telling him that he is about to be attacked and if it is extremely likely that the message will be intercepted because security has been breached, then this might lead to catastrophic attacks against both you and your neighbor. This has an even more negative benefit than would doing nothing; in one case both of you get attacked, in the other case only the neighbor gets attacked. In such a situation the service request should be rejected.

In summary, the decision theoretic approach allows AWDRAT to account for variations in the client's preference, variations in the availability of resources, and variations in the assessment of the trustability and reliability of computational resources. It allows AWDRAT to consider explicitly whether to attempt recovery at a particular time and whether to attempt to render a service at all. It represents a cognitive, self-aware approach to making the application systems immune to compromises of the resources.

1.2.4 Vulnerability Analysis and Attack Plan Recognition

Attack modeling is the process of systematically enumerating all of the ways in which a computational environment can be attacked and discovering how those attacks can lead to resource compromises. The output of attack modeling is a set of complex, multi-step plans that an attacker might use. These plans are then interpreted by the plan recognition component of the system which is informed by inputs from the full gamut of sensors available. It collates these inputs looking for specific patterns of activity characteristic of each step of a plan; for example, high rates of password scanning alarms from an intrusion detection system are characteristic of an early stage of an attack in which the attacker is attempting to gain first access. The plan recognizer maintains a set of active hypotheses; each hypothesis corresponds to a particular attack plan some of whose steps have already been observed.

The attack plans are hierarchical; each sub-plan corresponds to a attack effecting a compromise enabling other sub-plans downstream. As we described in section 1.2.1, the bottom tier of the trust model deals with attacks while the middle tier deals with the health status of resources. To reflect the causal relationship between attacks and compromised resource states, a Bayesian network node representing a sub-plan of an attack plan is linked to a node representing a compromised modes of one or more resources in the trust model. As the plan recognizer gathers evidence that an attack is in progress, the Bayesian machinery increases the estimate of the probability that the resource has been compromised, possibly causing this estimate to become large enough to warrant action. At such a point the system poses for itself a goal of removing the compromise, tries to find a method relevant to this recovery and if successful executes the method and changes its estimate that the resource is compromised. [10, 11].

Attack plans are generate by a rule-based inference system that uses the ontology underlying the trust model to reason about how one might affect a desirable system property. Fundamentally, this rule base deals with how different components depend on and control one another. We make this rule base as abstract and general as possible. This puts the notion of control and dependency at the center of the reasoning process. There are several rules about how to gain control of components, which are quite general. The following are examples of such general and abstract rules:

If the goal is to affect the reliable-performance property of some component ?x
Then find a component ?y of ?x that contributes to the delivery of that property
and find a way to control ?y

If the goal is to control a component ?x
Then find an input ?y to ?x and find a way to modify ?y

If the goal is to control a component ?x

Then find a component ?y of ?x and find a way to control ?y

If the goal is to control a component ?x
Then find a vulnerability ?y of the component ?x
  and find a way to exploit ?y to take control of ?x.

At the leaves of this reasoning chain is specific information about vulnerabilities and how to exploit them. For example:

- Microsoft IIS web-servers below a certain patch level are vulnerable to buffer overflow attacks.

- Buffer overflow attacks are capable of taking control of the components that are vulnerable.

Vulnerability analysis is a backward chaining goal-directed reasoning process. It begins with desirable properties, finds ways to compromise those resources that deliver these properties and then find ways to enable these compromises. The attack plans developed may be quite complex, multi-stage plans in which one step enables a compromise (e.g. gaining access to a use account) that serves as a foothold for succeeding steps (e.g. monitoring network traffic to steal information). Figure 5 gives an example.

```
To compromise the privacy of a typical computer C-1 in Cluster-1
    Know the contents of a typical file F-1 on C-1
      To do that Achieve Access rights to F-1
          To do that Know the password of a typical user U-1 in Cluster-1
              To do that Observe network traffic on the Cluster-1's Subnet S-1
                  To do that Control A Switch SW-1 connected to S-1
                      To do that Logon to SW-1
                         To do that Know the password for the administrators of SW-1
                            To do that launch a password guessing attack
                        and connect to SW-1 using the SSH protocol
and use those rights to Read F-1
```

Figure 5: A Plan for Affecting Privacy output by the Attack Modeler

1.2.5 Architectural Differencing

So far, we have discussed the trust model, how it is connected to attack recognition and diagnosis services (1.2.1), and how AWDRAT makes decisions about how to render application services given the information in the trust model (1.2.3). However, it is possible and unfortunately likely that some attacks will not be recognized and that resources will be compromised. Thus, for the overall system to be survivable, the trust model must be kept current while application software is running, the consequences of inappropriate behavior due to compromised resources must be contained, and the software must be reconfigured dynamically to avoid further inappropriate use of compromised resources. The key to all this is for AWDRAT to notice the misbehavior of the running application software, for it to use this as evidence of a compromised resource, for it to learn more about the nature of the compromise, and for it to reconfigure itself so as to reduce future risk.

Our approach is based on detecting deviations of a application's actual behavior from its expected behavior by running the executable application software in parallel with a simulation model of its intended behavior. The simulation model is supplied with input, output and state data gathered during execution using wrapper technology. A difference between the observed behavior of the application and that predicted by the model is taken to be symptomatic of an underlying problem and is used to trigger diagnostic services (1.2.7) whose job it is to characterize and localize the breakdown. We call this approach "Architectural Differencing".

In Section 1.2.2 we described the basic execution cycle of AWDRAT as centering around *metho*d *selection*. The method ultimately selected is, in fact, just normal executable code (including legacy code); however, we associate with each executable method a simulation model. This is expressed in an architectural description language, called SDSL (Statecharts for Dynamic Systems Language) [13] which draws on UML [34] we also draw on ideas from the Plan Calculus of the Programmer's Apprentice project [30].

The model is a coarse decomposition of the task into partially ordered sub-tasks connected by abstract data and control flow links. Each sub-task is annotated with prerequisite conditions, post-conditions, and invariant conditions that must hold throughout the execution of the sub-task as well as descriptions of state-changes and other dynamic system behavior that should occur during the method's execution.

The model is usually not elaborated down to the individual subroutine level of the software, it stays at a more abstract level of decomposition. However, the entry and exit points of each model sub-task are associated with points in the code at which wrappers may be interposed, most often at actual subroutine entry and exit points. Wrappers are also used to capture significant events during the execution of the method (e.g. object creation, communication between objects). Wrappers are implemented in a variety of ways depending on what the environment offers. In a suitably reflective language (e.g. Lisp) they may be implemented by employing the Meta-level of the language system. In other cases wrappers may be implemented by incorporating them into the source code of the system; if the source code is not available, (e.g., commercial of the shelf components -COTS) wrappers may be injected [12].

As long as AWDRAT can monitor at least the entry and exit points, it can maintain a mapping between the behavior of the executing application and that of the simulation model. It achieves synchronization between the two by translating significant execution events into inputs to the simulation. When a module in the executing application is started with particular inputs, these are presented to the simulation model which then makes predictions about: (1) What significant events should be noticed during the execution (and with what delay), (2) What outputs should be produced satisfying what constraints (both temporal and logical) and (3) What events should not be observed because they violate invariant conditions. As events in the executing software are noted, AWDRAT checks these for equivalence with the model's predictions, translating the raw events into the model's more abstract language and guaranteeing equivalence at the design level. Any difference between the predictions of the model and the observed behavior is treated as a symptom that is brought to the attention of the diagnostic services (1.2.7).

Although the Model representation specifies prerequisite, invariant and post-conditions for each module, it is not the case that all of these events are easily observed or that the constraints are computationally inexpensive to check. Thus, the model must include annotations specifying which conditions actually to check; monitoring points are only enabled for such tractably checkable conditions. However, the other conditions are still useful during diagnosis as will be seen in section 1.2.7.

We also include models of misbehavior, when these are known or anticipated. We decompose the models along aspectual lines, separately describing how the task might perform correctly or incorrectly from the point of view of performance (e.g, overly slow or fast execution), privacy (e.g. storing protected data in public places), etc. This allows us to conveniently characterize misbehaviors of various types without combinatorial explosion of the descriptions. These characterizations of incorrect behavior are employed during diagnosis, see 1.2.7.

The final set of annotations included in the model deal with critical data sets. As we will see in the section 1.2.6, we use wrappers to mediate access to all critical data sets. These wrappers enforce and signal violation of role-based access rules, initiating diagnostic activity. When indicated by the model, the wrappers can provision backup copies, checkpoints and access journals for all data marked as critical in the model. These can be used to aid diagnosis by helping to determine who or what processes accessed or modified the data; in addition they can be used during recovery to reconstitute compromised resources as will be described in the section 1.2.8.

1.2.6 Wrapper Synthesis

AWDRAT requires information to be collected during execution of an application method to determine whether the constraints of the application model are met. Moreover, critical data must be logged at certain points as execution progresses in case restoration to a safe state at a later time is necessary. To collect this data, we have chosen not to interfere with existing coding practices; we will not insist on trusting only code that is written anew for our purposes. Because of this, we cannot assume, for example, that we could modify the existing source code and recompile versions better suited to our modeling goals, perhaps broadcasting the data we need to collect. Instead, AWDRAT instruments existing legacy code with wrappers that collect the data without interfering with the original functionality.

At Teknowledge we have built a wrapper-based technology called Mediated Connectors that can be imposed on legacy components (1) To observe behavior that feeds AWDRAT's Architecture Differencer; (2) To create periodic backup copies of critical data resources; (3) To observe non-function properties, such as resource consumption and utilization. (4) In effecting recovery procedures from backed up data and other modeled trust data.

The Mediated Connector technology [3] is quite mature and has been used extensively (in several projects [14, 41, 42]); the technology is immune to mediator bypass attacks. The Mediated Connector technology is based on intercepting (mediating) calls to PC Windows-based platforms' Dynamic Link Libraries (DLLs). One or more sets of mediators are designed for each library to be wrapped; the mediators are programmed in C++. Each wrapped call has access to the parameters of the intercepted call and can substitute its own computation's result for the wrapped function's result. The wrapper can (and normally does) invoke the original function, possibly with altered parameters, in order to determine the result to pass back. After installation, the wrappers run in the calling process and have access to everything in its environment, e.g., its process id and spawning process.

Wrappers are normally used for three distinct purposes. First, they can be used to protect resources by checking to see that a critical function, e.g. “open file for output,” is not invoked on sensitive system resources, e.g. “files in the Windows Startup directory.” In this case, the result returned by the wrapper tends to be either an empty result or the wrapper actually throws an exception. Second, they can be used to observe and record activities in the system, e.g. “website access frequency,” by simply keeping auxiliary data structures or files that they update. Such wrappers keep their statistics or logs and allow the original function to proceed normally. A related use for wrappers is to cache results and return the same result as the first call on identical parameter sets. And finally, wrappers can be used to change functionality in part or in its entirety. One could spoof access results, for example, if the invoker of the accessor were not trusted.

Generally, this final activity requires cooperation among mediators to effect a new infrastructure. For example, one can impose an encryption decryption protection scheme using wrappers that cooperate on both storage and retrieval of information on a particular resource. Here we are proposing just such a new infrastructure for trust management.

To establish this infrastructure there will be some wrappers that we design that will be imposed on every system participant. These will be used to establish the reflective information model and are quite complex. For example, all process creation and deletion activity must be wrapped to maintain a reflective model of the running architectural structure of the system.

There are two approaches to synthesizing the model-specific wrappers needed to collect system-specific data used in modeling. One is to preprogram most wrappers for generic activities such as file access, registry access, and web access, as well as a second set of wrappers around process creation and destruction, etc. These wrappers would then interpret of more specific instructions relevant to their specific context of user. Although these mediators can be dynamically installed and activated, they would be largely pre-programmed. A second approach is to dynamically synthesize the specific wrappers needed to monitor specific predicates. The decision of which approach to take depends on the overhead of using the interpreted version and how volatility of the situation driving the synthesis process. Initially, we will work with the former approach and look for bottlenecks indicating the desirability of the latter. In either case, our Teknowledge group has over 25 years’ experience with program synthesis systems and can apply a variety of program generation technologies to this problem [43, 44, 45, 15, 46].

1.2.7 Diagnosis

We have described how we model computational behavior (1.2.5) and how the models act as active monitors of the executing software (1.2.6). As an application computation proceeds AWDRAT checks that the executing software produces behavior equivalent to that specified in its model. If there is a discrepancy, then the executing system is halted and diagnostic services are invoked.

The task of diagnosis is to localize and characterize the breakdown; in this case this ultimately comes down to (1) Identifying resources that have been compromised so as to cause the computation to misbehave in the manner observed and (2) Identifying those tasks that have been correctly completed and those conditions that can be relied upon as a basis for recovery. If every condition in the model were easily observed and efficiently checked, this would be trivial since AWDRAT would catch any misbehavior immediately. Unfortunately, this is usually not the case; the observable and verifiable behaviors are sparse and act as containment regions on the execution. The diagnostic task then becomes to employ model-based, symbolic inference to isolate and characterize the failure given the sparse observations that have been made. Finally, since there may be more than a single possible explanation for a failure (for example, more than one resource compromise might lead to the same misbehavior), AWDRAT employs Bayesian inference techniques to associate a probability with each possible explanation and ultimately to update the estimates in its trust model.

The techniques we employ were developed in the OASIS program [36, 38, 37] and draw on earlier model-based diagnosis work [7, 8, 6, 16]. The basic idea is as follow: Associated with the executing software are simulation models of both nominal and abnormal modes of behavior (1.2.5) for each sub-task. To allow for the fact that we cannot possibly model all faults, we include a “null model” for each task, representing all other behaviors. The diagnostic task is to assign one such mode of behavior to each sub-task, such that the predictions made by the simulation model are consistent with the observations of the actual behavior.

Like all Model Based Diagnosis techniques, ours is driven by the use of a network of justifications and dependencies (i.e. a Truth Maintenance Systems) built during model simulation. As the coordinated execution of the actual software and the simulation model proceeds AWDRAT maintains a database of assertions corresponding to the model conditions. Those model conditions that are actually observed and checked are marked as premises. Prerequisite conditions of a task are justified with links to those post-conditions of prior tasks that collectively entail the precondition. Each sub-task in the model has several possible behavioral modes, each with its own simulation model, prerequisite and post-conditions. Up to the point that a discrepancy is noticed, we assume that each sub-task executes in its normal mode. The post-conditions asserted by this model are linked to the assumption that the task is in its normal mode and to the prerequisite conditions of the task. Thus, as execution proceeds we build up a dependency network linking unobservable conditions to observations and the assumptions that sub-tasks behaved normally.

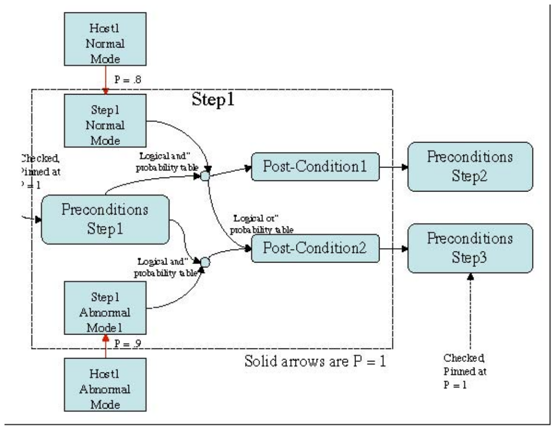


Figure 6: Dependencies Built During Diagnosis

Diagnosis is initiated when a discrepancy is detected between the expected and actual behaviors of a computation. We first identify the *conflic*t *set*; this is the set of components that were assumed to be behaving normally and that are linked by justification chains to predicted behavior that is at variance with the observed behavior. Diagnosis consists of finding different behavior modes for some or all of the components in the conflict set such that the selected behavior models predict the behavior actually observed (as opposed to that predicted by the normal behavioral modes). There may in fact be several such diagnoses. As we search for diagnoses, we may also encounter selections of abnormal behavior models that make predictions at variance with the actual observations. These are additional conflict sets. We continue enumerating combinations of behavioral models until we have identified all the conflict sets; all remaining combinations are possible diagnoses. This whole process is managed efficiently by a truth maintenance system as described in [38, 37, 7, 8, 9, 39].

As we noted in sections 1.2.1 and 1.2.5, the trust model links different behavioral modes for tasks in the model to the various kinds of compromises that might be present in the resources employed by the computation. In fact, these are all represented within a Bayesian inference network. The Bayesian network is augmented with a new node corresponding to each conflict set; each such conflict node is connected to the nodes corresponding to behavior modes in the conflict set and the conditional probability table of this node corresponds to a logical conjunction (i.e. its true value has probability 1 if all the inputs are true and otherwise the true value has probability 0). Since the conflict set is a set of behavior modes that collectively entail conditions at variance with our observations, we pin the value of each such node at false.

Actually, we only identify minimal conflicts (i.e. conflict sets that are not supersets of other conflicts) since these imply any non-minimal conflicts. After all minimal conflicts are discovered, any remaining set of behavioral modes is a consistent diagnosis. For each of these we create a node in the Bayesian network which is the logical-and of the nodes corresponding to the behavioral modes of the components. This node represents the probability of this particular diagnosis; the probability of this node is determined by the Bayesian inference machinery. Finally all the dependency relationships between prerequisite conditions, selection of behavior mode and post-conditions are also added to the Bayesian network[4] .

The Bayesian network is then solved giving us updated (posterior) probabilities. In particular, we have posterior probabilities for each diagnosis, for each possible compromised mode of each resource, and for each assertion in our trace of the computation that was not directly observed. Since these assertions represent prerequisite and post-conditions of tasks in the model, we also have estimates of how likely it was that these particular conditions obtained at particular points in the computation.

In summary, our approach to diagnosis uses model-based symbolic reasoning as well as Bayesian probabilistic reasoning to assess the probabilities that resources have been compromised in specific ways; this is to say that diagnosis updates the trust model. In addition, these same diagnostic techniques assess the probabilities of the various way in which the computation might have misbehaved so as to produce the misbehavior observed. This is to say that diagnosis updates a world model, telling us what conditions we can rely on even after the computation has failed (and to what extent we can trust this assessment). These are then used to support AWDRAT's recovery and regeneration processes.

1.2.8 Recovery from Failure and System Regeneration

Having reached a diagnosis of a failure and updated the trust model, AWDRAT attempts to find an alternative way to render the desired service. It also considers the broader question of regenerating other resources that are suspect and that may be required for system wide consistency.

In the final analysis, all recovery and regeneration efforts are a matter of replacing corrupted data by more trusted copies. However, by data we mean any collection of bits in either volatile or non-volatile storage that is used in the course of a computation. There are several aspects to this problem. First, the same conceptual data may exist in several locations and move between these during normal operation. For example, a segment of executable code might exist in primary memory as well as in a paging partition. Secondly, the same conceptual data might exist in different formats (e.g. source code, binary files, linked image files) that are transformed one into another. In general, it may be possible to achieve short term recovery by fixing the active version alone, but systematic regeneration would fix all versions and representations that are suspect.

AWDRAT's primary recovery technique is to attempt to replace the corrupted data (including code) representing a computational resource by a redundant copy that had

[4] we will explore an implementation technique in which the data structures representing justifications in the Truth Maintenance system also represent links in the Bayesian network, removing the need to maintain what is essentially redundant data

been secreted away by wrappers[5]. Once the resource is restored, its status in the trust model is updated to reflect its highly increased likelihood of being uncorrupted.

2. Detailed Description of Key Technologies

The AWDRAT architecture is shown in Figure 1. AWDRAT is provided with models of the intended behavior of its applications. These models are based on a "plan level" decomposition that provides invariant conditions for each module's execution as well as for the modules' pre-and post-conditions. AWDRAT actively enforces these declarative models of intended behavior using "wrapper" technology. Non-bypassable wrappers check the model conditions at runtime, allowing execution to proceed only if the observed behavior is consistent with the model's constraints. We call this technique "Architectural Differencing". In the event that unanticipated behavior is detected, AWDRAT uses Model-Based Diagnosis to determine the possible ways in which the system could have been compromised so as to produce the observed discrepancy. AWDRAT proceeds to use the results of the diagnosis to update a "trust model" indicating the likelihood and types of compromise that may have been effected to each computational resource. Finally, AWDRAT helps the application recover from failure, using this trust model to guide its selection of computational techniques (assuming that the application has more than one method for carrying out its intended tasks) and in its selection of computational resources to be used in completing the task.

AWDRAT uses its model of an application's intended behavior to recognize the critical data that must be preserved in case of failure. AWDRAT generates wrappers that dynamically provision backup copies and redundant encodings of this critical data. During recovery efforts, AWDRAT installs these backup copies in place of compromised data resources.

AWDRAT, using this combination of technologies, provides "cognitive immunity" to both intentional and accidental compromises. An application that runs within the AWDRAT environment appears to be self-aware, knowing its plans and goals; it actively checks that its behavior is consistent with its goals and provisions resources for recovery from future failures. AWDRAT builds a "trust model" shared by all application software, indicating which resources can be relied on for which purposes. This allows an application to make rational choices about how to achieve its goals in light of the trust model.

2.1 Wrapping Technology

Wrappers are used to allow AWDRAT to gain visibility in the program's execution. AWDRAT, in fact, employs two distinct wrapper technologies: SafeFamily[4, 19] and

[5] Our project did not attempt to develop sophisticated techniques for managing and protecting this backup data; in the future we will attempt to use the techniques developed by other projects in SRS.

JavaWrap. The first of these encapsulates system DLL’s, allowing AWDRAT to monitor any access to external resources such as files or communication ports. The second of these provides method wrappers for Java programs, providing a capability similar to “:around” methods in the Common-Lisp Object System[21, 5] or in Aspect-J[22].

These two capabilities are complementary: JavaWrap provides visibility to all application level code, SafeFamily provides visibility to operations that take place below the abstraction barrier of the Java Language runtime model.

Together they provide AWDRAT with the ability to monitor the applications behavior in detail as is shown in Figure 7.

Both wrapper technology’s involve the use of a collection of individual mediators each of which gains control at the entry point of some module. For JavaWrap the unit of granularity is the individual Java method, for SafeFamily, the unit of granularity is the individual API entry point in a shared library.

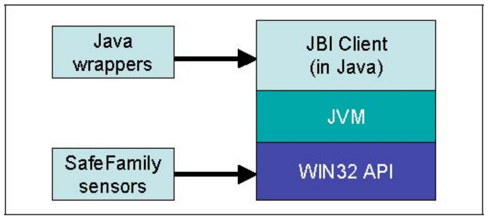


Figure 7: Two Types of Wrappers Used in AWDRAT

A mediator typically does one or more of the following:

- conditionally calls the mediated API
- calls the mediated API with altered actual parameter values
- return a different result (or exception) than the mediated API

Mediating one or more APIs from a shared library provides, (or one or more method in a class file) in effect, a new library (class file) that relies on the original. The new library (or class file) exports the same interface as the original, and actually shares with the original the binary code that is common to both. Mediator authors are thus relieved of the need to reimplement any portions of the library they don’t need to mediate. They are also relieved of any need to maintain a second copy of the source (generally unavailable anyway). Mediator users do not need to retain a second binary version of the library (class file) in the file system. The new library (class file) is virtual. It is implemented dynamically within the address space of any process that uses the mediators.

2.1.1 The SafeFamily Facility

The SafeFamily facility is build on top of the low level mediators that are used to gain control of the API's presented by the shared libraries (dll's) that constitute the OS interface. The SafeFamily execution environment controls the set of resources that can be seen by application code running in the mediated environment. It also controls how and to what extent those resources can be used. Any excluded resources are neither detectable nor useable by the applications running within these execution environments.

By restricting how resources are used, these SafeFamily execution environment can prevent the modification or destruction of critical resources, access of unauthorized information, submission of unauthorized requests, and denial of service.

The SafeFamily execution environment protects four classes of resources: files, the system registry, process spawning, and communication with remote hosts. Access to, and modification of, each of these resources is controlled by a set of user specified rules. The rules are written in XML notation, with one unit for each Resource. To use the SafeFamily facility, one must provide an XML file of rules specifying the resources (e.g. files, ports) and actions (e.g. writing the file, communicating over the port) that are to be prevented.

The Figure 8 shows a rule that controls files in the "AWDRAT" directory:

```
<file inherit="true" override="false"
            resource="C:\aire2\edu\mit\aire\awdrat\" specialmode="none">
    <read action="allow" audit="false"/>
    <write action="allow" audit="false"/>
    <execute action="deny" audit="true"/>
    <com action="deny" audit="true"/>
  </file>
```

Figure 8: A SafeFamily Rule

The SafeFamily security manager non-bypassably installs wrappers on newly spawned processes as they are started in accordance with the wrapper policy specification it is given. This wrapper policy specification details which wrappers are to be placed on which processes whenever those processes run.

This Security Manager runs as a service (which means that it is started before any user logs on) and monitors all process spawns. It accomplishes this monitoring by wrapping all the other services and any processes they spawn (recursively). This includes all processes started on behalf of the logged in user. The Security Manager also propagates all wrappers installed on a process to all processes that that process spawns directly or indirectly.

Within the AWDRAT environment, any policy violation detected by the SafeFamily facility results in the action being blocked and a message being sent on a socket opened between the SafeFamily security manager and the AWDRAT executive. AWDRAT inserts an event corresponding to this signal into its event stream, allowing the execution monitor to respond to the attempt at unauthorized access. We have a designed, but not yet implemented, a more complex mechanism in which AWDRAT can respond to the SafeFamily security manager, telling it that the attempted access is, in fact, allowable in the current context. In more detail, the SafeFamily executive blocks the thread that attempted the unauthorized access, and then signals the AWDRAT executive (which runs in a separate thread from the application). The AWDRAT executive then gets to determine whether this access is one that allowed at this point in the execution of the program. If so, it signals back to SafeFamily security manager, that the access is permitted and then the SafeFamily manager allows that application thread to continue execution. If AWDRAT determines that the access is, in fact, a violation, then it signals that back to the SafeFamily security manager which continues to block the access.

Thus, the SafeFamily rules act as a coarse grained policy, saying which accesses to resources are never allowed and which are conditionally allowed, while the AWDRAT execution monitor supplies the fain-grained control based on its system model.

2.1.2 The JavaWrap Facility

Just as the SafeFamily wrapping facility provide visibility at the shared library API level (which is ideal for monitoring resource accesses), so the JavaWrap facility provide visibility into the dynamic execution of the application code at the granularity of individual method calls. To use the JavaWrap facility, one must provide an XML file specifying the methods one wants to wrap as well as a Java Class of mediator methods, one for each wrapped method in the original application. This is explained in more detail below.

JavaWrap is a tool that installs mediators on Java methods in a Java application. A mediator is itself a Java method that mimics the interface of the method that it mediates (the mediatee). Technically, a mediator replaces its mediatee in the wrapped application. The mediator has access to the mediatee, to its parameters, and (in the case of a non-static mediatee) to the this object supplied at the invocation of the mediatee. A wrapper consists of a collection of mediators and methods called by them. A wrapper generally also contains its own variables in which its mediators record aspects of the application state.

The most general form of mediator is called a transformer. Transformers are the most common type of mediator used in AWDRAT. Although a transformer completely replaces the behavior of its mediatee, it can include in that behavior an invocation (or even more than one!) of the mediatee, passing either the original parameters or different parameters.

Two less general, but quite common, uses of mediators are singled out with simplified protocols for their definition. A monitor is a mediator that applies its mediatee to the parameters originally supplied in its invocation, returning the value (if any) returned by the mediatee, but performs some additional behavior (typically recording information in the wrapper’s state) before and/or after the execution of the mediatee. Typically the additional behavior consists of recording state information in the wrapper’s private variables. This additional behavior can include actions that modify the application’s native state, but monitors are not intended for that purpose. Monitors are used in AWDRAT mainly for gaining visibility into invocations of Java constructor methods.

An authorizer is a mediator that decides whether or not to allow an invocation of its mediatee to be executed. The authorizer has access to the mediatee’s parameters and (in the case of a non-static mediatee) to the object supplied at the invocation of the mediatee. The authorizer may throw an exception to the caller, blocking execution of the mediatee, or may allow the mediatee invocation to occur as it would have had there been no mediator.

Monitors can be used to produce logs of application execution. JavaWrap includes a capability for producing fairly general logs, in an XML format, of an application’s execution. Using JavaWrap to produce logs requires you to identify which methods should be traced, and where the log should be written, but does not require you to write any mediators.

2.1.2.1 Implementation and Requirements

Java Wrappers are implemented as transformations to the byte code implementations of Java classes. The transformations to a class are carried out at the time the class is loaded into the Java Virtual Machine (JVM).

Sun Microsystem’s JVM version 1.5 (aka 5.0) includes a facility for java agents to intervene in the loading of classes. Java wrappers rely on this facility and so will not work with earlier versions of the JVM. Only the 1.5 JRE (Runtime Environment) is required, not the full JDK. It can be freely downloaded from http://java.sun.com/j2se/1.5.0/download.jsp. Use of Java Wrappers does not require familiarity with java agents.

Java Wrappers also relies on a “Java source to byte code” compiler provided by the Javassist library. This library can be downloaded by following the link from http://www.jboss.org/products/javassist.h Use of Java Wrappers does not require familiarity with Javassist.

2.1.2.2 Wrapping a Java Application

Wrapping a Java application requires creating an XML file – the wrapper specification – and creating a java jar file – the wrapper implementation. The wrapper specification binds methods to be mediated to their mediators. The wrapper implementation contains the byte code implementations of the mediators.

To run the application with the wrapper, it must be launched with a -javaagent argument supplied to the JVM, and with a class path enhanced to include the wrapper implementation jar file and the Javassist jar file. For example, suppose an unwrapped application could be launched with the command line

{Java -cp <path> <mainclass> [args...]}

The same application could be launched with a wrapper using the command line:

{Java -javaagent:<JWjar>=<JWSpec> -cp <path>;<JWImpl>;<JAjar> <mainclass> [args...]}

where

- **<**JWjar> is the path to JavaWrap.jar
- **<**JWSpec> is the path to the wrapper specification XML file
- **<**JWImpl> is the path to the wrapper implementation jar file
- **<**JAjar> is the path to Javassist.jar

2.1.2.3 Wrapper Specifications

A Wrapper Specification is an XML document whose root element uses the tag JAVAWRAP. The immediate children of the JAVAWRAP root element use the tag CLASS. A CLASS element should have a single attribute, whose name is name and whose value is the fully qualified Java name of the class, using the “dot-separated” syntax you would use in java source code -e.g.,

{<CLASS name="org.xml.sax.Locator"> }

The name in a CLASS element must identify a Java class, not an interface.

A CLASS element may have any number of METHOD children and any number of PRINTER children. A METHOD element identifies a method of its parent’s class that is to be mediated or traced. The target method is identified by a pair of attributes: name and signature. The value of the name attribute is the simple name of the method, just as you would write it in a method invocation. The value of the signature attribute is the Java method descriptor for the method, using the notation specified in The Java™ Virtual Machine Specification (sections 4.3.2, 4.3.3, table 4.2),

The identified method may be either local to the class or, if it is an instance method, inherited from a superclass.

The method identified by a METHOD element, regardless of whether it is local or inherited, may not be an abstract method.

A constructor is identified in a METHOD element either by the name=”¡init¿” or by omitting the name attribute entirely.

```
{<METHOD signature= "(Ljava/lang/String;Z)V" />}
```

identifies a constructor with two parameters -a string and a boolean. If the class has only one method having the specified name -including inherited methods -then the signature may be omitted.

The method identified by the combination of the METHOD name and signature attributes, in conjunction with the CLASS name attribute, may be either an instance method or a static method, and may have any scope attribute (public, protected, private, or default).

If the method is to be mediated by code you supply, then the METHOD element must contain exactly one of the attributes: monitor, authorizer, or transformer. The value of the attribute should be a string that identifies the mediator to be used. The mediator code must be located in the Wrapper Implementation jar file. The identifying string consists of the mediator's full class name (dot-separated) followed by another dot and the method name e.g,

```
<METHOD    signature= "(Ljava/lang/String;Z)V" monitor=
        "tek.mafMed.Mediators.constructMission" />
```

All mediators must be static methods. The precise parameter types, return type, and allowed behavior of a mediator depends both on the method it mediates and on whether it is used as a monitor, authorizer, or transformer. These requirements are detailed below.

If the method is to be traced, then the METHOD element must contain none of the attributes: monitor, authorizer, or transformer.

2.2 Diagnostic Technology

AWDRAT's diagnostic service is described in more detail in [36] and draws heavily on ideas in [9]. Each component in the System Architectural Model provided to AWDRAT is provided with behavioral specifications for both its normal mode of behavior as well as additional models for faulty behavior. As explained in the section on Architectural Differencing, page 29, an event stream, tracing the execution of the application system, is passed to the execution monitor, which in turn checks that these events are consistent with the System Architectural Model. As the execution monitor does this, it builds up a data base of assertions describing the system's execution and connects these assertions in a dependency network. Any directly observed condition is justified as a "premise" while those assertions derived by inference are linked by justifications to the assertions they depend upon. In particular, post-conditions of any component are justified as depending on the assumption that the component has executed normally as is shown in Figure 9. This is similar to the reasoning techniques in [35].

Should a discrepancy between actual and intended behavior be detected, this will show up as a contradiction in the database of assertions describing the application's execution history. Diagnosis then consists of finding alternative behavior models for some subset of the components in the architectural model such that the contradiction disappears when these models of off-nominal behavior are substituted.

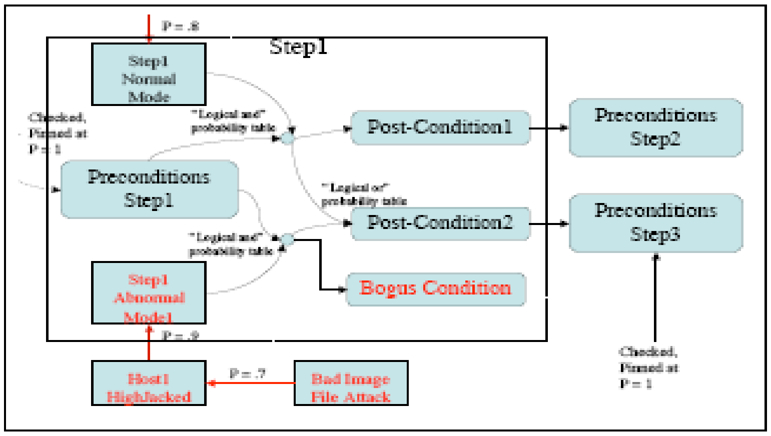


Figure 9: Dependency Graph

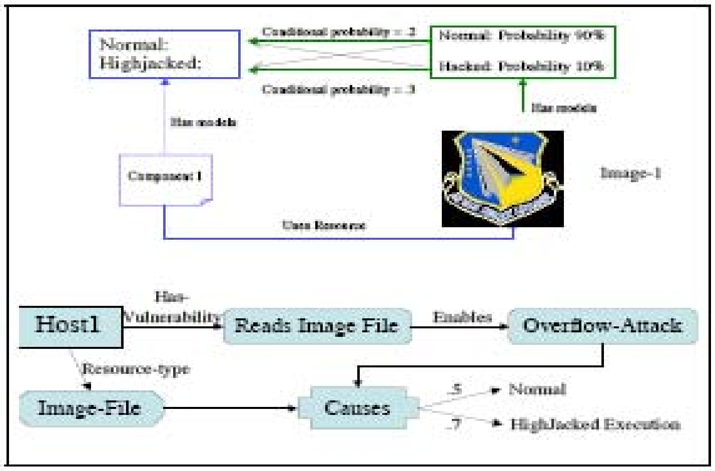


Figure 10: Diagnosis With Fault and Attack Models

2.2.1 Common Mode Failures

What we are able to observe is the progress of a computation; but the computation is itself just an abstraction. What an attacker can actually affect is something physical: the file representing the stored version of a program, the bits in main memory representing the running program, or other programs (such as the operating system) whose services are employed by the monitored application.

Thus, we require a more elaborated modeling framework detailing how the behavior of a computation is related to the state of the resources that it uses. In addition to modeling the behavior of the components in the system architectural model, AWDRAT therefore also models the health status of resources used by the application. We use the term "resource" quite generally to include data read by the application, loadable files (e.g. Class files) and even the binary representation of the code in memory. In turn, we must represent the vulnerabilities of these resources and the attacks enabled by these vulnerabilities. Finally, we must represent how such attacks compromise the resources, causing them to behave in an undesired manner.

The System Architectural Model provided to AWDRAT describes how a compromise to a resource might result in an abnormal behavior in a component of the computation; these are provided as conditional probability links. Similarly, AWDRAT's general knowledge base contains descriptions of how various types of attacks might result in compromises to the resources used by the application as is shown in Figure 10.

A single compromise of an operating system component, such as the scheduler, can lead to anomalous behavior in several application components. This is an example of a *commo*n *mod*e *failure*; intuitively, a common mode failure occurs when a single fault (e.g. an inaccurate power supply), leads to faults at several observable points in the systems (e.g. several transistors misbehave because their biasing power is incorrect). Another example comes from reliability studies of nuclear power plants where it was observed that the catastrophic failure of a turbine blade could sever several pipes as it flies off, leading to multiple cooling fluid leaks.

Formally, there is a common mode failure whenever the probabilities of the failure modes of two (or more) components are dependent. Early model-based diagnostic systems have assumed probabilistic independence of the behavior modes of different components [9] in order to simplify the assessment of posterior probabilities. Later work [39] allows for probabilistic dependence; however, it does not explore in detail how to model the causes of this dependence. We deal with common mode failures by extending our modeling framework to make explicit the mechanisms that couple the failure probabilites of different components.

As already noted we have extended our modeling framework, as shown in Figure 10, to include two kinds of objects: computational components (represented by a set of delay models one for each behavioral mode) and infrastructural components (represented by a set of modes, but no delay or other behavioral models). Connecting these two kinds of models are conditional probability links; each such link states how likely a particular behavioral mode of a computational component would be if the infrastructural component that supports that component were in a particular one of its modes (normal or abnormal). Each infrastructural component mode will usually project conditional probability links to more than one computational component behavioral mode, allowing us to say that normal behavior has some probability of being exhibited even if the infrastructural component has been compromised.

The model also includes a *prior*i probabilities for the modes of the infrastructural components, representing our best estimates of the degree of compromise in each such piece of infrastructure. Following a session of diagnostic reasoning, these probabilities may be updated to the value of the *posterio*r probabilities.

We next observe that resources are compromised by attacks. Attacks are enabled by vulnerabilities in the resources. For example, many systems in the Unix family are vulnerable to buffer-overflow attacks; most networked systems are vulnerable to packet-flood attacks. An attack is capable of compromising a resource in a variety of ways; for example, buffer overflow attacks are used both to gain control of a specific resource and to gain root access to the entire system. But the variety of compromises enabled by an attack are not equally likely (some are much more difficult than others). We therefore add a third tier to our model to describe the ensemble of attacks assumed to be available in the environment. We connect the attack layer to the resource layer with Conditional probability links that state the likelikhood of each mode of the compromised resource once the attack has been successful.

When attacks are present in the environment what matters is the conditional probabilities of the different modes of the resources given that an attack has taken place. We hypothesize that one or more attack types are present in the environment, leading to a three-tiered model as shown in figure 10.

Our model of the computational environment therefore includes:

- The components of the computation that is being observed

- A set of behavioral models for each component, representing both normal and failure modes.

- The set of resources available to be used by the computational components

- A set of behavioral modes for each resource, representing both normal and compromised modes.

- A map stating which resources are used by each computational component.

- Conditional probabilities linking the modes of the computations to the modes of the resources employed by that component.

- A list of vulnerabilities possessed by each computational resource

- A description of which attacks are enable by each vulnerability.

- A list of attack types that are believed to be active in the environment.

- A description of which compromised modes of each type of resource can be caused by a successful execution of each type of attack. This is provided as a set of conditional probabilities of the compromised mode given the execution of the attack.

Given this information, simple rule-based inferencing (implemented in the Joshua inference system) deduces which specific resources might have been compromised and with what probability. This information is then used to construct a Bayesian network (in the IDEAL system).

2.2.2 Diagnostic Reasoning

As in earlier techniques, diagnosis is initiated when a discrepancy is detected; in this case this means that the predicted production time of an output differs from those actually observed after an input has been presented. The goal of the diagnostic process is to infer as much as possible about where the computation failed (so that we may recover from the failure) and about what parts of the infrastructure may be compromised (so that we can avoid using them again until corrective action is taken). We are therefore looking for two things: the most likely explanation(s) of the observed discrepancies and updated probabilities for the modes of the infrastructural components.

To do this we use techniques similar to [9, 39]. We first identify all conflict sets, and then proceed to calculate the posterior probabilities of the modes of each of the computational components. We do these tasks by a mixture of symbolic and Bayesian techniques; symbolic model-based reasoning is used to predict the behavior of the system, given an assumed set of behavioral modes. Whenever the symbolic reasoning process discovers a conflict (an incompatible set of behavioral modes), it adds to the Bayesian network a new node corresponding to the conflict (see below). Bayesian techniques are then used to solve the extended network to get updated probabilities.

This approach involves an exhaustive enumeration of the combinations of the models of the computational components. This allows us to calculate the exact posterior probabilities. However, this is expensive and the precision may not be needed. It would be possible to instead use the techniques in [40] to generate only the most likely diagnoses and to use these to estimate the posterior probabilities; but we have not yet pursued this approach.

We instead follow the following approach: We alternate the finding of conflicts with the search for diagnoses. After each “conflict” node is added to the Bayesian network (see below) the network is solved; this gives us updated probabilities for each behavioral mode of each component. We can, therefore, examine the behavioral modes in the current conflict and pick that component whose current behavioral mode is least likely. We discard this mode, and pick the most likely alternative; we continue this process of detecting conflicts, discarding the least likely model in the conflict and picking its most likely alternative until a consistent set is found. This process is a good heuristic for finding the most likely diagnosis [6].

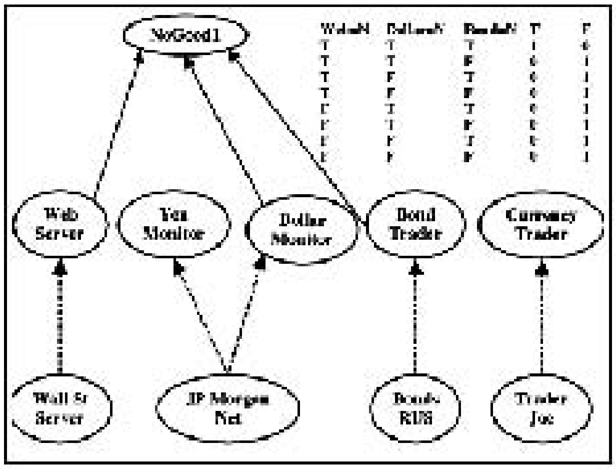


Figure 11: Adding a Conflict Node to the Bayesian Network

Our models of computational behavior are used to predict the behavior of the computational components and to compare the predictions with observations. When a discrepancy is detected, we use dependency tracing to find the conflict set underlying the discrepancy (i.e. a set of behavioral modes which are inconsistent). At this point a new (binary truth value) node is added to the Bayesian network representing the conflict as shown in Figure 11 (based on a fictional financial system). This node has an incoming arc from every node that participates in the conflict. It has a conditional probability table corresponding to a pure ”logical and” i.e. its true state has a probability of 1.0 if all the incoming nodes are in their true states and it otherwise has probability 1.0 of being in its false state.

Since this node represents a logical contradiction, it is pinned in its false state. Adding this node to the network imposes a logical constraint on the probabilistic Bayesian network; the constraint imposed is that the conflict discovered by the symbolic, model-based behavioral simulation is impossible. We continue to explore other combinations of behavioral modes, until all possible minimal conflicts are discovered. Each of these conflicts extends the Bayesian network as before. The set of such conflicts constitutes the full set of logical constraints on the values taken on within the Bayesian network; thus,

6 However since the probabilities of the failure modes of different components are not independent, this is only a heuristic

once we have augmented the Bayesian network with nodes corresponding to each conflict, the network has all the information available.[7]

At this point, we have found all the minimal conflicts and added conflict nodes to the Bayesian network for each. We therefore also know all the possible diagnoses since these are sets of behavioral modes (one for each component) which are not supersets of any conflict set. For each of these we create a node in the Bayesian network which is the logical-and of the nodes corresponding to the behavioral modes of the components. This node repre
sents the probability of this particular diagnosis. The Bayesian network is then solved. This gives us updated probabilities for all possible diagnoses, for the behavioral modes of the computational components and for the modes of the underlying infrastructural components. Furthermore, these updated probabilities are those which are consistent with all the constraints we can obtain from the behavioral models. Thus, they represent as complete an assessment as is possible of the state of compromise in the infrastructure. These posterior estimates can be taken as priors in further diagnostic tasks and they can also be used as a "trust model" informing users of the system (including self adaptive computations) of the trustworthiness of the various pieces of infrastructure which they will need to use.

2.3 Architectural Differencing

In addition to synthesizing wrappers, the AWDRAT generator also synthesizes an "execution monitor" corresponding to the system model as shown in Figure 15. The role of the wrappers is to create an "event stream" tracing the execution of the application. The role of the execution monitor is to interpret the event stream against the specification of the System Architectural Model and to detect any differences between the two as shown in Figure 12. Should a deviation be detected, diagnosis and recovery is attempted. Our diagnosis and recovery systems, far and away the most complex parts of the AWDRAT run-time system, are written in Common-Lisp; therefore, the actual "plumbing" generated consists of Java wrappers that are merely stubs invoking Lisp mediators that, in turn, signal events to the execution monitor, which is also written in Lisp. This is shown in Figure 13.

The architectural model provided to AWDRAT includes prerequisite and post-conditions for each of its components. A special subset of the predicates used to describe these conditions is built-in to AWDRAT and provide a simple abstract model of data structuring. The AWDRAT synthesizer analyzes these statements and generates code in the Lisp mediators that creates backup copies of those data-structures manipulated by the application that the architectural model indicates are crucial.

[7] [39] builds logical reasoning directly into the Bayesian network system because the logical inferences needed are simple enough to be accommodated. However, our inference needs are more complex and not easily amenable to this approach

The execution monitor behaves as follows: Initially all components of the System Architectural Model are inactive. When the application system starts up it creates a "startup" event for the top level component of the model and this component is put into its "running" state. When a module enters the "running" state it instantiate its sub-network (if it has one) and propagate input data along data flow links and passes control along control flow links. When data arrives at the input port of a component, the execution monitor checks to see if all the required data is now available; if so, the execution monitor checks the preconditions of this component and if they succeed, it marks the component as "ready". Should these checks fail, diagnosis is initiated. As events arrive form the wrappers, each is checked. If the event is a "method entry" event, then the execution monitor checks to see if this event is the initiating event of a component in the "ready" state; if so, the component's state is changed to "running". Data in the event is captured and applied to the input ports of the component. If the event is a "method exit" then the execution monitor checks to see if this is the terminating event of a "running" module; if so, it changes the state of the component to "completed". Data in the event is captured and applied to the output ports of the component. The component's post-conditions are checked and diagnosis is invoked if the check fails. Otherwise the event is checked to see if its an allowable or prohibited event of some running component; detection of an explicitly prohibited event initiates diagnosis as does the detection of an unexpected event, i.e. one that is neither an initiating event of a ready component, or a terminating or allowable event of a running component. Using these generated capabilities, AWDRAT will detect any deviation of the application from the abstract behavior specified in its System Architectural Model and AWDRAT's diagnostic services will be invoked.

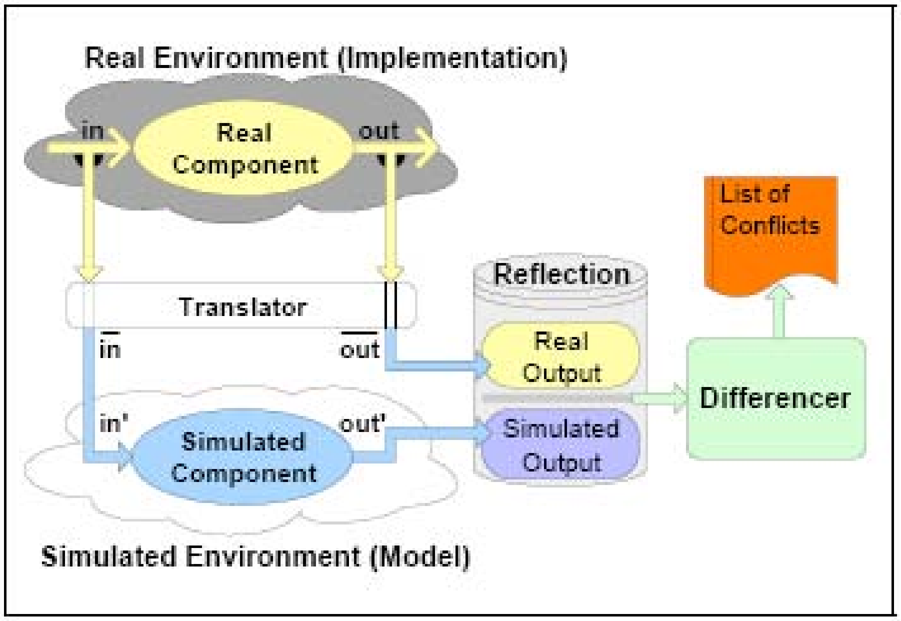


Figure 12: Architectual Differencing

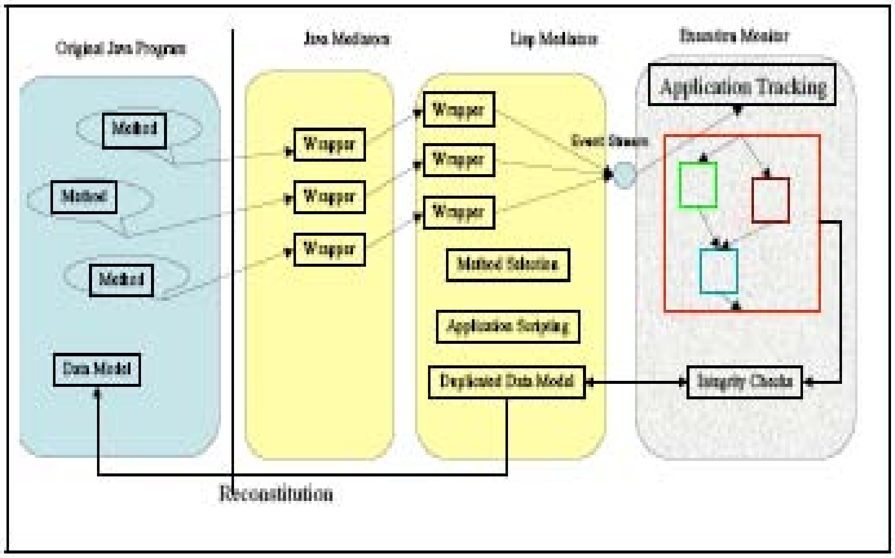


Figure 13: The Generated Plumbing

2.4 Self-Adaptive Software

Recovery in AWDRAT depends critically on self-adaptive techniques such as those described in [24]. The critical idea is that in many cases an application may have more than one way to perform a task. For example, in the experiments that will be described in the section on experimental methods, 3, we tethered a graphical editor application to AWDRAT. This application loads image files (e.g. GIF, JPEG) and, as it happens, there is a vulnerability (since fixed) related to loading malformed image files. This is enabled by the use of a "native library" (i.e. code written in 3); however, there is also a slower pure-java library that performs the same task.

Self-adaptive software involves making dynamic choices between alternative methods such as the native and Pure Java image loading methods. The general framework starts from the observation that we can regard alternative methods as different means for achieving the same goal. But the choice between methods will result in different values of the "nonfunctional properties" of the goal; for example, different methods for loading images have different speeds and different resulting image quality. The application designer presumably has some preferences over these properties and we have developed techniques for turning these preferences into a utility function representing the benefit to the application of achieving the goal with a specific set of non-functional properties. Each alternative method also requires a set of resources consistent with constraints peculiar to that method and we may also think about these resources having a cost. As is shown in Figure 2, the task of AWDRAT's adaptive software facility is to pick that method and set of resources that will deliver the highest net benefit. Thus AWDRAT's self-adaptive software service provides a decision theoretic framework for choosing between alternative methods.

2.5 Recovery and Trust Modeling

Recovery in AWDRAT depends critically on self-adaptive techniques such as those described in [24]. The critical idea is that in many cases an application may have more than one way to perform a task. For example, in the experiments that will be described in the section on experimental methods, page 3.3, we tethered a graphical editor application to AWDRAT. This application loads image files (e.g. GIF, JPEG) and, as it happens, there is a vulnerability (since fixed) related to loading malformed image files. This is enabled by the use of a "native library" (i.e. code written in 3); however, there is also a slower pure-java library that performs the same task.

Self-adaptive software involves making dynamic choices between alternative methods such as the native and Pure Java image loading methods. The general framework starts from the observation that we can regard alternative methods as different means for achieving the same goal. But the choice between methods will result in different values of the “nonfunctional properties” of the goal; for example, different methods for loading images have different speeds and different resulting image quality. The application designer presumably has some preferences over these properties and we have developed techniques for turning these preferences into a utility function representing the benefit to the application of achieving the goal with a specific set of non-functional properties. Each alternative method also requires a set of resources consistent with constraints peculiar to that method and we may also think about these resources having a cost. As is shown in Figure 2, the task of AWDRAT’s adaptive software facility is to pick that method and set of resources that will deliver the highest net benefit. Thus AWDRAT’s self-adaptive software service provides a decision theoretic framework for choosing between alternative methods.

2.6 The Synthesizer

The inputs to these two wrapper generator facilities (the JavaWrap XML spec, the Java Mediator files and the SafeFamily XML specification file) are not provided by the user, but are instead automatically generated by AWDRAT from a “System Architectural Model” such as that shown in Figure 14. The model is written in a language similar to the “Plan Calculus” of the Programmer’s Apprentice [30, 35, 29]; it includes a hierarchical nesting of components, each with input and output ports connected by data and control-flow links. Each component is provided with prerequisite and post-conditions. In AWDRAT, we have extended this notation to include a variety of event specifications, where events include the entry to a method in the application, exit from a method or the attempt to perform an operation on an external resource (e.g. write to a file). Each component of the architectural model may be annotated with “entry events”, “exit events”, “allowable events” and “prohibited events”. Entry and exit events are described by method specifications (and are caught through the JavaWrap facility); allowable and prohibited events may be either method calls or resource access events (resource access events are caught by the SafeFamily facility). The occurrence of an entry (exit) event indicates that a method that corresponds to beginning of a component in the architectural model has started (completed) execution. Occurrence of a prohibited event is taken to mean that the application system has deviated from the specification of the model.

Given this information, the AWDRAT wrapper synthesizer collects up all event specifications used in the model and then synthesizes the wrapper method code and the two required XML specification files as is shown in Figure 15.

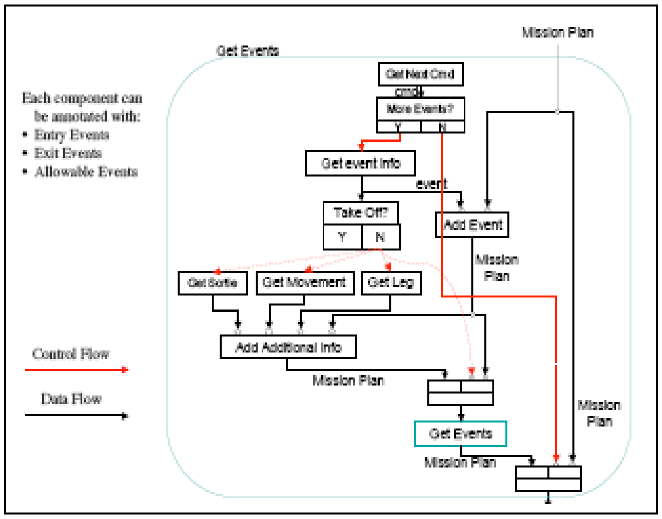


Figure 14 An Example System Model

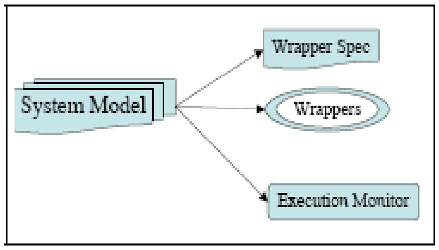


Figure 15: Generating the Wrapper Plumbling

3. Technical Results

AWDRAT's goal is to guarantee that the application tethered to it faithfully executes the intent of the software designer; for example, for an interactive system this means that the system should faithfully execute the commands specified by its user (e.g through an application GUI), or for a server application this means that it should faithfully execute the requests received from its client applications.

3.1 The Testbed System

3.1.1 Requirements

Our approach was predicated on applying operator commands and directives to an operational system model to predict the effect of those actions on the system, and to them guarantee that those effects and no others had been realized in the system. In addition, we wanted to be able to use our technology to capture and replicate the internal data structures of the program, so that we could restore the program to its previous state in the event that the program was corrupted. We thus needed a real system for which we could easily build an operational model, and which we could easily instrument.

The application system we selected was the MAF interactive mission planner – a component of the DARPA Dem/Val demonstration system which is in turn based on Rome Labs Joint Battlespace Infosphere (JBI). MAF is a Java-based program whose primary vulnerabilities arise either through use of native libraries containing unsafe code or through an attack that somehow manages to gain enough privileges to modify the application's class files.

3.1.2 OASIS Dem/Val: A Legacy System

To demonstrate the applicability of the AWDRAT architecture to legacy systems, we chose a moderately large example legacy system to model and defend against insider attacks, the OASIS Dem/Val system developed under an earlier DARPA program. This system relies on the Joint Battlespace Infosphere (JBI) repository and communication protocol for coordinating and managing information from a variety of agents cooperating in the development of major military plans. The OASIS Dem/Val system developed air tasking orders for air cargo and munitions delivery and deployment and was created to demonstrate how existing military systems could interoperate with new components through the JBI infrastructure.

Most of the agents in the Dem/Val scenario were programmatic "stub code" that published pre-canned information from files, rather than, for example, publishing actual weather data. However, a substantial operator interface illustrated in Figure 17 was provided for the MAF and CAF agents (the same component, actually, just applied to different information at different times for different purposes). . Notice the locations of various airfields around the world and the route being constructed from the US to Africa. These are easily changed by actions of the user before publishing the ATO, using the bottom right button at the left of the screen in the cluster of 5.

Because this subsystem was an actual application of modest complexity (several thousand lines of Java code) and because it has some vulnerabilities to outsider attacks, it was chosen as the component that the AWDRAT Architecture would protect.

3.1.2.1 OASIS Operational Scenario Description

The motivation and actual scenario chosen to demonstrate our technology using the OASIS system is best described by the operations manual itself [20].

As context for the OASIS scenario is Operation Allied Force, the NATO military operation fought primarily with air power and used to compel Serbia to cease hostilities against ethnic Albanians in Kosovo. This air operation allowed peace-keeping forces, on the ground, to carry out their mission to a successful conclusion. Much of the success in Kosovo was due to incredible efforts of the individuals involved in the planning and execution of operations, but a tool like the OASIS JBI would hopefully make their job easier.

Within this theater of operations, our scenario will be scoped to focus only on some of the functions performed by an AOC/TACC and its constituent planning cells. We will describe the separate planning processes that occur within the AOC/TACC in developing, refining, and executing an Air Battle Plan against WMD facilities, taking environmental factors such as weather and chemical plume hazards into consideration.

Table 1: OASIS Dem/Val Glossary of Terms

AOC	Air Operations Center	MIDB	Modernized Integrated Database
AODB	Air Operations Database	NATO	North Atlantic Treaty Organization
ATO	Air Tasking Order	OASIS	Organically Assured and Survivable Information Systems
CAF	Combat Air Forces	SPI	Sensor Performance Impact
EDC	Environmental Data Cube	TACC	Tanker Airlift Control Center
HTML	Hyper Text Markup Language	TAF	Terminal Aerodome Forecast
IO	Information Object	TAP	Theater Air Planner
IR	InfraRed	TBMCS	Theater Battle Management/Core Systems
JBI	Joint Battlespace Infosphere	TNL	Target Nomination List
JEES	Joint Environment Exploitation Segment	TWS	Theater Weather Server
JWIS	Joint Weather Impact System	USMTF	US Message Text Format
MAF	Mobility Air Forces	WMD	Weapons of Mass Destruction
METAR	METeorological Air Report	XML	eXtensible Markup Language

The associated use case models targeting and mission planning for air strikes against weapons of mass destruction (WMD) facilities. Weather and chemical plume/aerosol effects are taken into consideration during this mission. Weather changes affect predicted WMD plume dispersion, requiring the in-flight sortie to stand down in order to prevent undesirable propagation of the plume.

The process begins with the Targeting Cell in the Air Operations Center producing a Target Nomination List (TNL). The Combat Plans Cell then takes this TNL and builds the Air Tasking Order (ATO) assigning strike packages against each target. The ATO is then sent out to each unit/squadron who will be participating the strikes. A few hours after the TNL is distributed, the Combat Operations Cell comes online to start monitoring weather conditions, readying aircraft, and implementing the ATO when it is built.

Throughout this planning process there is opportunity to take weather effects into account. Weather affects such things as weapon/sensor head selection, route selection, and attack timing. In addition, when considering an attack against a WMD facility, we have to assess where the released chemical materials will travel and ensure that neither noncombatants nor friendly forces will be harmed.

While the ATO is being built, the TACC in St. Louis is also planning an in-theater mission. An airlift mission to Prince Sultan Air Base is built that involves in-air refueling in the North Sea and landings in Aviano and Sigonella. This flight will be reconciled with the in-theater Director of Mobility (DirMob) to ensure that weather conditions and other factors will permit the flight. Once the DirMob approves the mission, notice is sent back to the TACC MAF planner for execution.

The ATO is then finalized and distributed to Combat Operations, who is responsible for mission execution and monitoring. After planes have departed for their targets, updated weather conditions become available from Air Force Weather Agency (AFWA.) After analysis of this latest weather by the Chemical Hazard Cell, it is predicted that a change in winds around the WMD site will cause a toxic plume to encroach a heavily populated area of non-combatants. The new plume pattern is passed to Combat Operations, who redirects the WMD sortie to stand down and return to base. Table 1 should help to clear up the meanings of most of the acronyms.

The normal flow of publications through the system is shown in Figure 16. A solid arrow indicates publication by the agent at the root of the arrow and the dotted arrows represent receipt of the information by the agent at the tip of the arrow (no matter which direction the arrow is pointing). Time progresses downward, although some of the timings are artificial. For example, it does not matter whether WLC is published before or after WH, because their consumers are disjoint and neither produces a publication before both are received.

3.1.3 Instrumenting Dem/Val

Our focus in this project was on the use of AWDRAT 1) to detect malicious attacks on the MAF system, 2) to appropriately diagnose these attacks in terms of the characterizing the ways in which the system is corrupted and in terms of localizing the point of the attack, and 3) to restore the MAF system to a consistent state from which the user can continue useful work.

Doing this required modeling the internal flows of the MAF system, modeling the data structure invariants that apply to the internal state of the MAF system, modeling the set of events that are expected and allowed to occur within the execution of the MAF system. In addition, restoring the system to a consistent state required us to build a set of control routines that are capable of “scripting” the MAF system so as to replicate the effect of the user's interaction through the MAF system GUI. These were all effected through use of the key AWDRAT capabilities Architectural Differencing, Wrappers, Diagnosis, Recovery, Adaptive Software and Trust modeling

3.2 Red Team Exercise

The AWDRAT red teaming exercise was conducted on October 18 and 19, 2005 at the MIT CSAIL facility. The Red Team was from Raba Technologies LLC. The exercise ran all day, October 18, 2005 and was continued for the full morning of October 19. Prior to the actual red-team exercises, we met with the Red Team members from Raba Inc. on October 17,

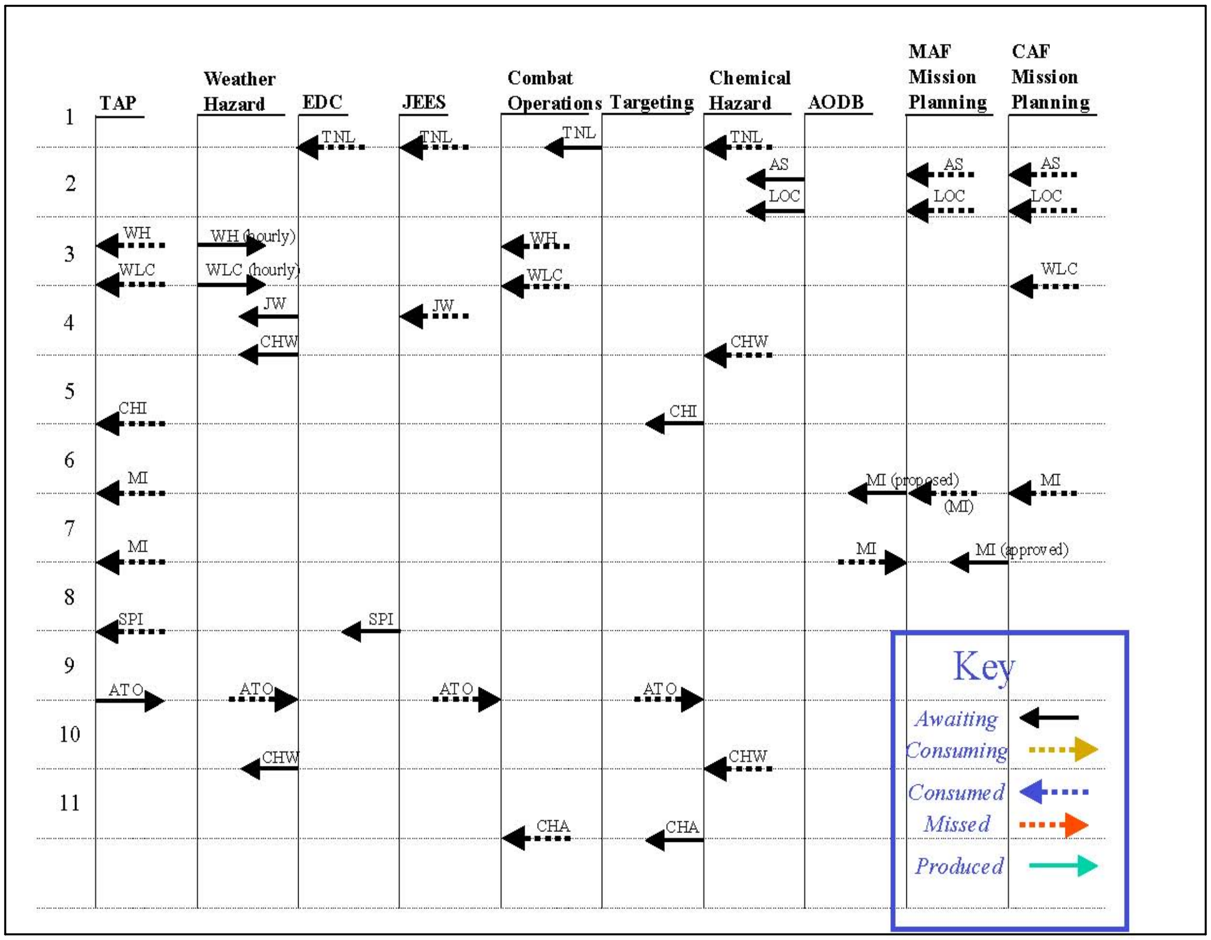


Figure 16: OASIS Dem/Val Scenario

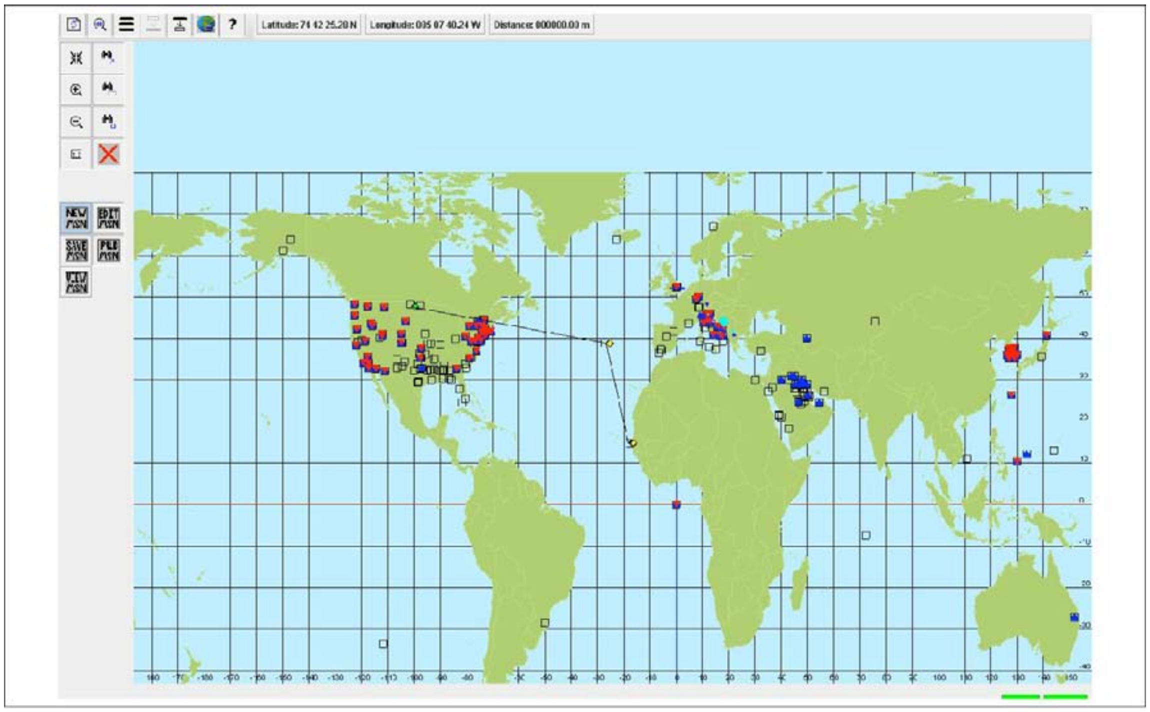


Figure 17: The MAF / CAF GUI

to describe the MAF application in detail and the goals of the AWDRAT architecture. In addition, RABA forwarded to us a draft Assessment Plan before out meeting.

Several things mitigated against the exercise being as informative as it might have been:

- The red-team and we did not spend adequate time interacting before the tests. Thus, there was significant disconnect between the parties in terms of the focus and significance of the tests. This exercise was one of the first Red-Team engagements in the SRS program and there wasn't enough of an experience base to lead both sides to a correct understanding of the necessary preparation, particularly preparation involving discussion between both team about the goals and methodology of the exercise. (In contrast, the PMOP exercise in which we also participated involved extensive prior interactions and was much more informative).

- To a significant degree, the Red-Team approached the exercise as a rather traditional perimeter penetration exercise. Thus, many of the tests they had prepared were actually outside the scope of the exercise. A measure of the problem, is to be seen in the fact that only 10 of the 25 different attacks, prepared by the red-team were considered in scope.

- When we tried to redirect the testing effort towards the issues that were of significance to AWDRAT, the only way to proceed was for Blue Team members to aid the Red Team in inserting "attack code" into the sources and recompiling.

The first couple of such attempts involved code that “exec’d” Windows Explorer, a illegal action that should have been discovered by the SafeFamily facilities, except that SafeFamily was incorrectly installed (see next point).

- We did not spend adequate time installing our software on the experimental machine nor did we adequately test that the installation was correct. As a result, the SafeFamily software was misconfigured and inoperable leading to a failure all of defenses that depended on the SafeFamily capability. In addition, there were some minor bugs in the core AWDRAT software (2, each requiring 1 line of code to be changed) there weren’t discovered until after the exercise finished.

Table 2: Red Team Results

Name	Value
Total Attacks Discussed	25
Attacks Deemed Out of Scope	9
Attacks Not Scored	4
Attacks Not Tested	3
Red Team Victory	8
Blue Team Victory	2

Faults identified	2	
Total Faults	10	
Percentage of faults identified	2/10	20%
Corrective action Percentage of corrective action	2 2/10	20%

None of the above discussion is intended to lay blame on the Red-Team. The fundamental problem was the lack of interaction between the parties with enough lead time before the exercise. Consequently there was a significant lack of mutual understanding about goals and methods. In addition, we credit the Red Team with a substantive evaluation of the AWDRAT system and for perceptive suggestions about future work. One addition point that seems clear from the experience is that it is often useful to stage a red-team exercise in two engagements, the first to sort out the rough points and the second to collect meaningful results.

The following data are drawn from the White Team report on the exercise, prepared by Chris Do.

The Red Team prepared twenty-five (25) different attacks that were supposed to be executed. In accord with the issues discussed earlier, and because of time constraints, only ten attacks were executed and scored. Eight were in favor of the Red Team and two for the Blue Team. Several of the issues settled in favor of the Red Team were issues that we conceded on the grounds that they were actually outside of our research goals and were clearly things that we did not defend against, but that nevertheless seemed important.

The SRS program goal was to accurately diagnose at least 10% of the root causes of system problems and to automatically take effective corrective action for at least half of those diagnoses. The results in table 2 show that AWDRAT performed better than the requirements of the program metric. Thus, in spite of several difficulties in the experiments, AWDRAT met the program goals in the Red Team experiments.

The next section discusses internal experimentation done after the Red Team exercise that stressed the defenses much more systematically than was done in the Red Team exercise.

3.3 Internal Experiments

AWDRAT's primary concern is not with the attack vector, but rather with the identification of and containment of misbehavior and with the identification of corrupted resources so that recovery processes can avoid them. In particular, AWDRAT focuses on the maintenance of data integrity and data privacy.

We identified the following properties that AWDRAT should guarantee:

- The data structures maintained by the application should accurately reflect user requests:
    - Each field of each data structure should be what the GUI requested
    - No other fields should be present
    - All the fields requested by the GUI should be present
- The application should not be able to open a port to any application other than to the JBI servers with which it must communicate (i.e. the JBOSS server).
- The application should not be able to open a port to a foreign host
- The application should not be able to write files except to the specified directories that constitute its temporary storage areas.

3.3.1 Experimental Methodology

To test the effectiveness of AWDRAT in guaranteeing these properties we adopted three distinct ways of attempting to violate the properties:

- Modification of class files. We hand inserted attack code into a Java source file, recompiled and placed the class file into the appropriate directory on the Java Classpath. This is intended to simulate the case where an external attacker gained sufficient privileges to corrupt the class files.

- Random attacks. The AWDRAT controller launches a parasitic thread on startup. This thread waits for a random delay and then attempts to violate one of the constraints listed above (e.g. attempts to launch an application, to open a port, or to corrupt the data structures). Which type of violation (and in the case of data integrity which field of which data structure) is chosen randomly; the specific attack selected is logged for analysis purposes.

- Wrapped methods. AWDRAT places wrappers around a significant number of methods in the MAF application. The wrapper can be used a place from which to launch a simulated attack; for example, by wrapping the "loadImage" method, one can simulate an attack payload that is carried by a corrupted image file (without actually having to do the very laborious work of constructing such a corrupted image file).

The core AWDRAT machinery is kept unaware of the attack machinery. Its goal is to detect and characterize a violation.

The MAF's data-structures are a relatively simple, tree built from instances of a few classes. The top level is a Mission-Builder containing a hashtable for the overall mission data and four hashtables holding the sets of events, legs, sorties and movements data structures. Each entry in these tables is an instance of the appropriate type, containing a hashtable with the data specific to that element of the mission plan. Each flight path (mission) is represented by a Mission-Builder that contains a set of Events, Legs, Sorties, Movements. An Event is a Take-off, a Landing, or a Waypoint. For each take-off event there is a corresponding leg, sortie and movement:

When considering data-structure integrity it is important to understand that the data structures can be modified using two different levels of calls. Each data-structure is implemented as a Java Class with accessor methods (e.g. setInformation, getInformation). In addition, Java, of course, defines methods on hash-tables (e,g, put, get). The application always accesses the data structures through its API using the getInformation and setInformation methods. However, payload code might access the data-structures below the level of this API (e.g. using the hash-table methods or even using lower-level, memory accessing capabilities at the Native Code level). Thus it is necessary to simulate attempts to corrupt the data structures using both the API methods and the hash-table methods (AWDRAT does not wrap or monitor the hash-table level methods for both pragmatic and technical reasons, so modifications at this level are as opaque as modifications using Native code level capabilities).

Each of the simulated attacks methods has drawbacks and strengths: The first method allows very precise placement of an attack within the code, but might be criticized for being overly synchronized to the normal execution flow of the program; the random attacks, has a dual set of issues. However, our feeling is that collectively, these tests cover a great deal of the appropriate space.

In more detail, here are the tests conducted by category:

1. Modification of Class Files:
- Insert call to write files
- Insert call to launch external application
- Insert call to open port
- Insert code to modify the data structures at the API level
- Insert code to modify the data structures below the API level

2. Random delay attacks (through a parallel thread)
- Make call to write files
- Make call to launch external application
- Make call to open port
- Attempt to modify the data structures at the API level
- Attempt to modify the data structures below the API level

3. Wrapped Methods (used only for loadImage)
- Make call to write files
- Make call to launch external application
- Make call to open port

The use of a wrapper to simulate an attack payload was only employed for the loadImage method (to avoid actually building a payload carrying corrupted image file). The wrapper only attempts to do the disallowed action when using the native loadImage library on specific image files identified in a configuration file; this is admittedly a bit hokey since the same wrapper mechanism is used both to launch the simulated attack and to make the decision as to whether to use the native or Pure Java load method, but we have tried to keep two separate and to run an honest test. In this case we do not attempt to corrupt the MAF data structures because loadImage is only called in the application during its initialization, which is before the data structures are created. So image based attacks only attempt to open a port or to write a file.

The second category of violation is launched from a thread that is started by the initialization code of the system. This thread waits until the user begins to enter a mission plan, then picks an arbitrary delay time (less than 4 minutes); after that delay time, it either attempts to open a port, write a file or to corrupt the data structures. To do the last of these, it picks an arbitrary element of the MAF data structures and attempts to either modify an existing field of the data structure, or to add a new field. Strictly speaking the later action is harmless, the application will ignore the extra field. However, the criterion for success is detecting any deviation of the application from the actions requested by the GUI, so we include these cases as well.

3.3.2 Detection methods

As explained in the sections on Wrappers, 2.1, Architectural Differencing, 2.3 and Diagnostic Reasoning, page 2.2, AWDRAT picks up violations in one of three ways: 1) It checks the integrity of the Java data structures against its internal backup copy everywhere that the system-model specifies that the data structures should be consistent. 2) It checks that monitored methods are called only at points in the execution sanctioned by the system model. 3) It receives messages from the SafeFamily (dll) wrappers, alerting it to violations of the access rules imposed by SafeFamily. Some violations that are conceptually in the same category (e.g. data structure integrity) are picked up by more than one mechanism. For example, an attempt to modify the MAF data structures using an API level call is usually picked up because the call isn't sanctioned at that point of the execution; however, using a hash-table method on a hash-table held in one of the data-structures will be picked up by the integrity check, since the hash-table methods aren't wrapped.

3.3.3 Results Summary

The results of out internal experiments are shown in figure 3. All attempts to launch an application, write a file other than those sanctioned or to open a port are detected. The only exception to this broad statement is that an attacker can write to a file in the MAF's temporary directory or open one of the ports used by AWDRAT itself (however, opening such a port would violate the rules of engagement for the experiments). Almost all attempts to destroy the integrity of the MAF data structures are detected; the exception is when the modification is made using the MAF API level calls during the execution of a method that legitimately uses the exact same API call. This almost only occurs in hand-modified source code "attacks" (one "random" attack managed to tickle this case). In principle, it's possible that an attack operating below the MAF API level could modify the MAF data structures and that the modification could be overwritten later by the uncorrupted MAF code doing the right thing. For example:

- GUI request that the "ALT" field of Event 1 be set to "30000"
- Attack code in another thread sets the "ALT" field of Event 1 to "1" using hash-table or lower level calls
- MAF method sets the "ALT" field of Event 1 to "30000"

The net effect is that the data structures are uncorrupted; however, AWDRAT's machinery will never detect the unsuccessful attempt to corrupt the data structures in this case.

3.3.4 Discussion

The first category includes hand inserted attack code. Four of these included calls to MAF API level methods inside other routines that legitimately invoke these methods.

These attacks could have been caught if AWDRAT had interposed itself at a higher level in the call tree, specifically at the boundary between the GUI and the rest of the MAF application. As it happens, the MAF system is not appropriately modularized to facilitate this; it could have been done, but it would have involved a bit more work. Of course, had AWDRAT interposed at that level, then it would be possible to imagine an attack interceding in the GUI code itself; this issue was discussed with the Red Team who felt that they would be unable to construct such an attack (with reasonable effort). One of the random attacks managed by pure chance to run in parallel with an application routine that invoked the same MAF level API; we thought this was too unlikely to actually occur, but we were wrong. This slips by because the current AWDRAT execution monitor isn't aware of thread structure: AWDRAT believes that the program is inside the method that builds an EVENT data structure; it believes this because it received an Enter Method event for that method. However this legitimate thread that generated the event isn't the "attack thread" and the AWDRAT execution monitor could use this information; this is an obvious next extension to AWDRAT.

Attacks that attempt to open ports, write files, or launch applications are intercepted by the SafeFamily wrapper system; the attempt is blocked so no bad effect is propagated. That is why the last column is marked Not Applicable (NA) for these categories of attack.

Table 3: Results of Internal Experimentation

Attack Type	Number of Attempts	Detected	Diagnosed	Corrected
Hand Placed	24	20/83%	20/83%	11/73%
Data MAF API	10	6/60%	6/60%	6/60%
Data lower API	5	5/100%	5/100%	5/100%
Port Opening	3	3/100%	3/100%	NA
File Write	3	3/100%	3/100%	NA
Application Launch	3	3/100%	3/100%	NA
Random	26	25/96%	25/96%	13/92%
Data MAF API	6	5/83%	5/83%	5/83%
Data lower API	8	8/100%	8/100%	8/100%
Port Opening	5	5/100%	5/100%	NA
File Write	4	4/100%	4/100%	NA
Application Launch	3	3/100%	3/100%	NA
Wrapped	9	9/100%	9/100%	9/100%
File Write	3	3/100%	3/100%	3/100%
Port Opening	3	3/100%	3/100%	3/100%
Application Launch	3	3/100%	3/100%	3/100%
Total	59	54/91%	54/91%	33/86%

In fact, AWDRAT does restart the application and rebuild its data structures in these cases as well. For the Wrapped cases (i.e. those involving simulated corrupt image files) the last column is listed because the dominant diagnostic hypothesis in those cases is that a payload was launched from the image loading method. In those case, switching to the Pure Java method and/or using a different format of the image file constitutes successful recovery. In these cases, we did not mark these as NA, since there was significant decision making in the recovery process. In the other cases, the dominant diagnostic hypothesis is that the class files (and/or core image) was corrupted; if there is a significant probability that the class files were corrupted, then the recovery process involves switching the class path to backup copies of the JAR files.

As a parenthetical note, we observe that there are a set of hand-coded attacks that we might have created but didn't that would have evaded detection. As we configured the system, SafeFamily wrappers block all read and/or write attempts except to a set of directories that are used by the MAF system. However, a spurious write to those directories would be allowed regardless of the call site. However, most of these exploits could be detected if the SafeFamily wrappers consulted the AWDRAT execution monitor to see if the program is executing a routine that is supposed to be doing I/O. We are looking into implementing this capability.

Finally we note that there are no false positives. This is to be expected if the system model is a reasonable abstraction of the program.

3.4. Meeting the Program Metrics

The SRS program goal was to accurately diagnose at least 10% of the root causes of system problems and to automatically take effective corrective action for at least half of those diagnoses.

The results in table 2 in section 3.2 show that AWDRAT performed better than the requirements of the program metric, detecting 20% of the attacks and mitigating against all of these.

The results in table 3 in section 3.3 show a much higher level of detection and correction: (91% detection and diagnosis and 86% correction).

Thus AWDRAT in both exercises exceeded the program metrics. The red team exercise was conducted with the SafeFamily facility installed incorrectly, leading to a serious degradation of detection capability. During the internal experimentation, the more systematic use of wrapping at both the language level (JavaWrap) and shared library level (SafeFamily) led to a much more complete detection regime.

Both sets of experiments were conducted under the assumption that the AWDRAT infrastructure was off limits. However, the systematic use of the SafeFamily facility to wrap all processes can make the resources used by the AWDRAT system virtually impervious to attack. The AWDRAT vision also includes the use of plan recognition technology to track the signs of possible attacks and to use the likelihood of possible attack together with diagnostic results to update the trust model. We believe that this added capability would significantly improve AWDRAT's ability to protect its own core resources.

AWDRAT's ability to protect an application stems from the systematic monitoring of the program's execution and presence of a system model, specifying what behavior is expected. The allow the AWDRAT diagnostic capabilities to localize and characterize the fault which, in turn, guides the recovery effort through use of a trust model indicating which resources are likely to have been compromised.

4. Conclusions

The experimental results indicate that AWDRAT has the ability to detect the symptoms of a variety of different types of attacks. AWDRAT's monitoring mechanisms allow it to sense several different types of deviation from expected behavior:

- **Sequencing Violations**: The system's control flow deviates from that specified in the system model.
- **Data Corruption**: The data structures of the system do not adhere to the constraints of the system model.
- **Communications Violations**: The application attempts to communicate with external entities in ways outside of those described in the system model.
- **Data Modification**: The application attempts to modify data in ways precluded by the system model.

Each of these types of deviation are detected using the instrumentation provided by the wrappers that AWDRAT synthesizes. Detection of any such violation institutes diagnosis and recovery. When the system model is more detailed, the instrumentation records very precise information about the application's execution, making the diagnostic task relatively easy and accurate. Weaker system models are easier to construct, but pick up less information and leave more room for diagnostic ambiguity. In either case, diagnosis leads to updated estimates of how trustable the system resources are and the information gathered by the instrumentation can be preserved to allow the application system to be restarted and restored to the consistent state that existed before the compromise was effected.

Given this backup information AWDRAT is capable of restarting the application system's execution, allowing it to complete its task. In addition, the trust model indicates that certain resources are less likely to be trustable than others. During the recovery process, these resources are avoided if possible. Generally this depends on two things:

- The existence of redundant resources (e.g. initial data sets, class files) that can be substituted for the initial ones
- The existence of alternative methods for achieving the task that avoid the use of the suspected resources

In our current experiments we have utilized both of these approaches. An example of the second approach is given by the case when AWDRAT believes that a "corrupted image file" attack has been used. In this case, it uses the alternative Pure Java image library which avoids the corrupted image file. An example of the first approach, is given by the case when AWDRAT believes that a class file has been corrupted. In that case, AWDRAT switches the class path to point to an alternative set of class files (in a JAR file). In both cases, reconstituting the original resource would be a simple next step in keeping with the "Self Regenerative" theme of SRS.

The AWDRAT architecture is a general purpose framework that can be adapted to a variety of application systems and domains of application. Each application involves a significant modeling effort, but when the architecture is applied to many applications within a domain, much of the modeling effort can be amortized over the entire ensemble of applications. We still need to explore in much greater detail the trade-offs between the specificity of the system model, the effort to produce the model and the utility of the added specificity in terms of diagnostic resolution and effectiveness of the recovery actions.

5. Key Personnel

In addition to the project's Principal Investigators, Howared Shrobe and Robert M. Balzer, several other researchers on their respective staffs at Massachusetts Institute of Technology and Teknowledge Corp. participated. Robert Ladaga of MIT was instrumental in the Common Lisp application development and algorithm development for the trust models and assessment. David Wile and Alexander Egyed built the infrastructure for coordinating and visualizing the activities of the various OASIS agents and the analyzers. Neil Goldman built the Java wrapper tool used to intercept the JBI services calls. Tim Hollebeek provided the wrapped Windows system call defenses to intercept malicious file and communications resource attacks. Marcelo Tallis wrote the JBI driver code and the detailed MAF / CAF GUI wrappers.

6. Comparison with Current Technology

The scope of AWDRAT is extraordinarily broad, encompassing fault containment, diagnosis, architectural modeling, wrapper technology, and self-adaptive software. We know of no other comparable overall effort. Nor do we know of other efforts that make a systematic attempt to build trust models, to derive attack plans using systematic reasoning from first principles, or that attempt to use Trust Models in a decision theoretic adaptive framework other than our own [38, 37, 36]. Our diagnostic techniques draw on a long tradition of model-based diagnosis in which we were early pioneers [7, 6, 8, 9, 16, 39] while the work on self-adaptivity is part of growing body of research [31, 32, 33, 23]. Our system modeling language is similar to other modern architectural description languages such as [17, 18, 25, 26, 27, 13].

The general spirit of our effort is similar to other research in the Information Survivability tradition which has focused on Intrusion Detection and Response. However, our approach to this is unique. There are three traditional approaches to intrusion detection [1] as shown in Figure 18. The first approach relies on a library of known attacks, finding behavior that matches such an attack pattern. A second approach is statistical in character; it builds a statistical profile of normal user interactions with the system and then detects behavior outside of this profile. A third approach uses a corpus of interactions annotated by an expert to indicate where attacks have occurred; a statistical recognizer is then trained using supervised machine learning techniques to recognize such attacks. The strengths and weaknesses of each of these traditional approaches can be measured in terms of their false positive and false negative rates. In addition, we may evaluate them by their diagnostic resolution: when the system signals an alarm, how precisely does it characterize the attack, the location of the attack and the nature of the compromise effected by the attack.

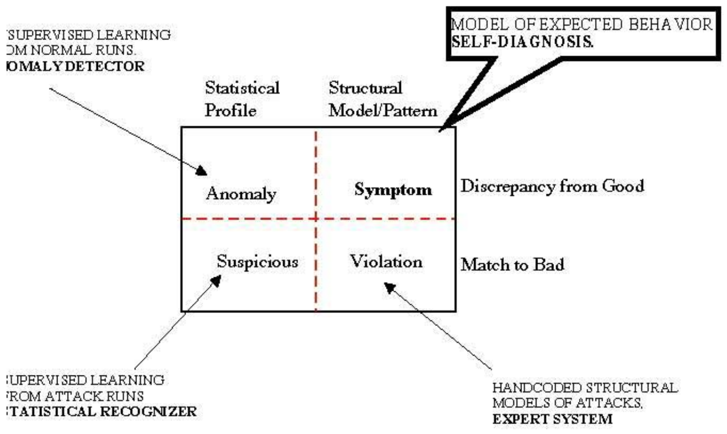
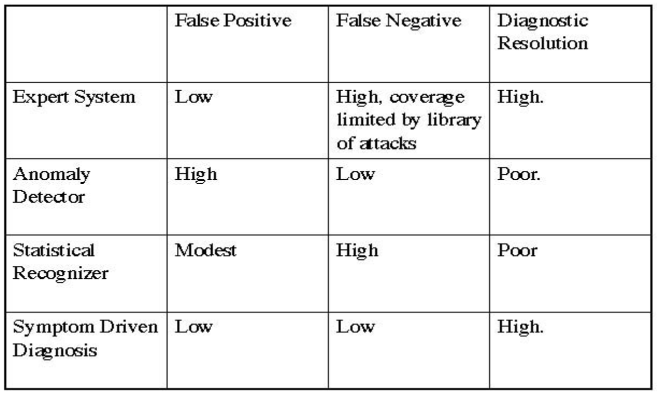


Figure 18: Approaches to Intrusion Detection



	False Positive	False Negative	Diagnostic Resolution
Expert System	Low	High, coverage limited by library of attacks	High.
Anomaly Detector	High	Low	Poor.
Statistical Recognizer	Modest	High	Poor
Symptom Driven Diagnosis	Low	Low	High.

Figure 19 Evaluation of Intrusion Detection Approaches

Figure 19 shows the strengths and weaknesses of these different approaches to intrusion detection. Expert systems are limited by the breadth of their library of known attacks. They have low false positive rates because they signal intrusions only when a precise pattern is matched; this also enables good diagnostic resolution. But they only recognize attacks in their knowledge base; in practice, the attackers are far more effective in generating new attacks and the knowledge base can't cope with the volatility of the environment. A high false negative rate is the result. In contrast, statistical anomaly detectors do not depend on recognizing attacks at all; their false negative rate is therefore much lower, since almost all attacks do fall outside the statistical profile of normal behavior. However, this approach lacks diagnostic resolution since all it knows is that the behavior is outside the profile. This approach conflates illegitimate behavior with unusual behavior leading to a high false positive rate; in practice, the profiles are never accurate enough to include many legitimate behaviors that are nevertheless infrequent. The machine learning approach, like the expert system approach is only as good as its training set; but keeping the training set up to date is nearly as difficult as keeping an expert system's knowledge base current. Consequently this approach suffers from a modest false negative rate; it also suffers from a modest to high false positive rate since the machine learning algorithm offers statistically generalizes to include legitimate behaviors. However, when such a system correctly recognizes an attack, it can usually provide some diagnostic resolution.

Finally, we use diagnosis to update the Trust Model which then acts indirectly in shaping the response. Any activities that are undertaken to repair compromised resources or to change the policies of boundary controllers are all evaluated within a common decision-theoretic framework. We believe that the great advantage of this approach is that we only undertake activities that seem to be "worth it," thereby avoiding self-inflicted denial of service problems.

7. References

[1] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers Univ., March 2000.

[2] R. Balzer. The dasada probe infrastructure. Technical Report Internal report (available from author), Teknowledge Corp.

[3] R. Balzer and N. Goldman.. Mediating connectors. In *Proceeding*s *o*f *th*e *19t*h *IEE*E *Internationa*l *Conferenc*e *o*n *Distribute*d *Computin*g *Systems*, pages 73–77. IEEE Computer Society Press, May 31-June 4 1999.

[4] Robert Balzer and Neil Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *Proceeding*s *o*f *th*e *Firs*t *Darp*a *Informatio*n *Securit*y *Conferenc*e *and* *Exhibitio*n *(DISCEX-II)*, volume II, pages 361–368, Jan 25-27 2000.

[5] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, , and D. Moon. Common lisp object system specification. Technical Report 88-002R, X3J13, June 1988.

[6] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificia*l *Intelligence*, 24:347–410, December 1984.

[7] Randall Davis and Howard Shrobe. Diagnosis based on structure and function. In *Proceeding*s *o*f *th*e *AAA*I *Nationa*l *Conferenc*e *o*n *Artificia*l *Intelligence*, pages 137–142. AAAI, 1982.

[8] Johan deKleer and Brian Williams. Diagnosing multiple faults. *Artificia*l *Intelligence*, 32(1):97–130, 1987.

[9] Johan deKleer and Brian Williams. Diagnosis with behavior modes. In *Proceeding*s *o*f *th*e *Internationa*l *Join*t *Conferenc*e *o*n *Artificia*l *Intelligence*, 1989.

[10] Jon Doyle, Isaac Kohone, William Long, Howard Shrobe, and Peter Szolovits. Agile monitoring for cyber defense. In *Proceeding*s *o*f *th*e *Secon*d *Darp*a *Informatio*n *Securit*y *Conferenc*e *and* *Exhibitio*n *(DISCEX-II)*. IEEE Computer Society, May 2001.

[11] Jon Doyle, Isaac Kohone, William Long, Howard Shrobe, and Peter Szolovits. Event recognition beyond signature and anomaly. In *Proceeding*s *o*f *th*e *Secon*d *IEE*E *Informatio*n *Assuranc*e *Workshop*. IEEE Computer Society, June 2001.

[12] A. Egyed and R. Balzer. Unfriendly cots integration -instrumentation and interfaces for improved plugability. In *Proceeding*s *o*f *th*e *16t*h *IEE*E *Internationa*l *Conferenc*e *o*n *Automate*d *Softwar*e *Engineerin*g *(ASE)*, November 2001.

[13] A. Egyed and D. Wile. Statechart simulator for modeling architectural dynamics. In *Proceedings of the 2nd IEEE/IFIP Conference on Software Architecture* (WICSA), pages 87–96, August 2001.

[14] N. Goldman. Smiley-an interactive tool for monitoring inter-module function calls. In *Proceedings of the 8th International Workshop on Program Comprehension*, June 10-11 2000.

[15] N. Goldman and R. Balzer. The isi visual design editor generator. In *Proceedings of IEEE Symposium on Visual Languages Conference*, Sept. 13-16 1999.

[16] Walter Hamscherand Randall Davis. Model-based reasoning: Troubleshooting. In Howard Shrobe, editor, *Exploring Artificial Intelligence*, pages 297–346. AAAI, 1988.

[17] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceedings of the 18th International Conference on Software Engineering*, pages 246–257, March 1996.

[18] D. Harel, H. Lachover, A. Naamad, A. Pnuell, M Poloti, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transaction on Software Engineering,* 16(4), 1990.

[19] Timothy Hollebeek and Rand Waltzman. The role of suspicion in model-based intrusion detection. In *Proceedings of the 2004 workshop on New security paradigms,* 2004.

[20] Doug Holzhauer and Carrie Kindler. Oasis scenario initialization and execution guide, April 2004.

[21] S. Keene. *Object-Oriented Programming in Common Lisp:* A *Programmer's Guide to CLOS.* Number ISBN 0-201-17589-4. Addison-Wesley, 1989.

[22] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming,* pages 327–353, 2001.

[23] R. Laddaga. Self-adaptive software sol baa 98-12. 1998.

[24] Robert Laddaga, Paul Robertson, and Howard E. Shrobe. Probabilistic dispatch, dynamic domain architecture, and self-adaptive software. In Robert Laddaga, Paul Robertson, and Howard Shrobe, editors, *Self-Adaptive Software,* pages 227–237. Springer-Verlag, 2001.

[25] D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering,* 1995.

[26] J. Magee. Behavioral analysis of software architecture using ltsa. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[27] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1996.

[28] Michael McGeachie and Jon Doyle. Efficient utility functions for ceteris paribus preferences. In *Eighteenth National Conference on Artificial Intelligence*. AAAI, July 2002.

[29] Charles Rich. Inspection methods in programming. Technical Report AI Lab Technical Report 604, MIT Artificial Intelligence Laboratory, 1981.

[30] Charles Rich and Howard E. Shrobe. Initial report on a lisp programmer's apprentice. Technical Report Technical Report 354, MIT Artificial Intelligence Laboratory, December 1976.

[31] P. Robertson. A *Self-Adaptive Architecture for Image Understanding*. PhD thesis, University of Oxford, 2001.

[32] P. Robertson, R. Laddaga, and H. Shrobe. *Self-Adaptive Software*. Springer-Verlag, 2000.

[33] P. Robertson, R. Laddaga, and H. Shrobe. *Self-Adaptive Software: Applications*. Springer-Verlag, 2002.

[34] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[35] Howard Shrobe. Dependency directed reasoning for complex program understanding. Technical Report AI Lab Technical Report 503, MIT Artificial Intelligence Laboratory, April 1979.

[36] Howard Shrobe. Model-based diagnosis for information survivability. In Robert Laddaga, Paul Robertson, and Howard Shrobe, editors, *Self-Adaptive Software*. Springer-Verlag, 2001.

[37] Howard Shrobe. Computational vulnerability analysis for information survivability. In *Innovative Applications of Artificial Intelligence*. AAAI, July 2002.

[38] Howard Shrobe. Model-based diagnosis for information survivability. In *Proceeedings of the International Workshop on Principles of Diagnosis, DX-02, http://www.dbai.tuwien.ac.at/user/dx2002/*, May 2002.

[39] Sampath Srinivas. Modeling techinques and algorithms for probablistic model-based diagnosis and repair. Technical Report STAN-CS-TR-95-1553, Stanford University, Stanford, CA, July 1995.

[40] Sampath Srinivas and Pandurang Nayak. Efficient enumeration of instantiations in bayesian networks. In *Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI-96)*, pages 500–508, Portland, Oregon, 1996.

[41] M. Tallis and R. Balzer. Document integrity through mediated interfaces. In *Proceedings of the Second DARPA Information Survivability Conference and Exposition (DISCEX II)*, June 2001.

[42] M. Tallis, N. Goldman, and R. Balzer. The briefing associate: A role for cots applications in the semantic web. In *International Semantic Web Working Symposium (SWWS)*, July 30-August 1 2001.

[43] D. Wile. Popart: Producers of parsers and related tools, reference manual. Technical report, USC/Information Sciences Institute, 1993.

[44] D. Wile. *Toward a Calculus for Abstract Syntax Trees.* Chapman and Hall, 1997.

[45] D. Wile. The lessons of the 80's. In *Proceedings of a Workshop on Transformation Technology, ICSE '99*, May 1999.

[46] D. Wile. Supporting the dsl spectrum. In *Journal of Computing and Information Technology CIT 9*, number 4, pages 263–287, 2001.

Appendix A

The System Model

```
(in-package :mediators)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Class Registry
;;; Defining short cuts for the class names
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(eval-when (:compile-toplevel :execute :load-toplevel)

    ;; Pad Editor Related Stuff

    ;; do we actually use this?

    (register-class "Pad" "mil.af.afrl.ife.gui.padeditor")

    (register-class "PadAction" "mil.af.afrl.ife.gui.padeditor")

    (register-class "PadEditor" "mil.af.afrl.ife.gui.padeditor")

    (register-class "PadDrawObjectListener" "mil.af.afrl.ife.gui.padeditor")

    (register-class "CMAPI" "mil.af.afrl.ife.cmapi.cmapi2k")



    ;; Frames and Panels
    (register-class "ClientFrame" "mil.af.rl.jbi.client.ExtensibleMappingClient.client")
    (register-class "ClientPanel" "mil.af.rl.jbi.client.ExtensibleMappingClient.client")
    (register-class "LoginWindow" "mil.af.rl.jbi.client.ExtensibleMappingClient.client")


    ;; Mission Objects


    (register-class "MissionObject" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "MissionBuilder" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan"
    :proxy '(mission-builder-proxy mission-holder-mixin))
    (register-class "MissionEventObject" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan"
    :proxy '(mission-event-proxy mission-event-mixin))
    (register-class "MissionLegObject" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan"
    :proxy '(mission-leg-proxy mission-stuff-mixin))
    (register-class "MissionMovementObject" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan"
    :proxy '(mission-movement-proxy mission-stuff-mixin))
    (register-class "MissionSortieObject" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan"
    :proxy '(mission-sortie-proxy mission-stuff-mixin))
    (register-class "MissionPlannerListener" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "MissionLegEditor" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "MissionViewerListener" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "MissionForm" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "MissionInfoForm" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "CreateMissionAction" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "EditMissionAction" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "PublishMissionAction" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "ReceiveMissionAction" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "SaveMissionAction" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "RequestHandler" "mil.af.rl.jbi.client.ExtensibleMappingClient.toolsets.MissionPlan")
    (register-class "mil.af.afrl.ife.gui.util.Util" "")
    (register-class "mil.af.afrl.ife.cmapi.util.Util" "")
    )


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Registering Events
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(register-event 'startup "LoginWindow"
"main"
'(("String[]" "args"))
:static t :startup t)


(register-event 'post-validate
"LoginWindow" "postValidate"
'(("String" "userName") ("String"
"pathname")))
(register-event 'create-client-frame "ClientFrame" "<init>"
 '(("String" "login")
    ("String" "password")))

(register-event 'exit "ClientFrame" "exitAction"
'()
)


(register-event 'center-action "ClientPanel" "centerAction"
'(("Point" "point")))


;;; Mission Event Object


(register-event 'create-mission-event-object "MissionEventObject" "<init>"
'())


    (register-event 'meo-set-information "MissionEventObject" "setInformation"
      '(("String" "key")

          ("String" "value")))

;;; Mission Builder

(register-event 'create-mission-builder "MissionBuilder" "<init>"

'())

(register-event 'create-mission-builder-with-client-panel "MissionBuilder" "<init>"

 '(("ClientPanel" "cp")))

(register-event 'create-mission-builder-with-hash-table "MissionBuilder" "<init>"

 '(("Hashtable" "hashTable")))

(register-event 'set-Initial-Info "MissionBuilder" "setInitialInfo"

 '(("Hashtable" "hashTable" )))

(register-event 'mission-builder-action-Performed "MissionBuilder" "actionPerformed"

 '(("ActionEvent" "actionEvent")))
```

```
(register-event ’mission-builder-set-Information "MissionBuilder" "setInformation"
 ’(("String" "Key")
   ("String" "Value")))

(register-event ’update-Msn-Evt "MissionBuilder" "updateMsnEvt"
’(("String" "Key")
   ("MissionEventObject" "missionEventObject")))

(register-event ’create-New-Additional-Mission-Info-Panel "MissionBuilder"
"createNewAdditionalMissionInfoPanel"
’(("Vector" "vector"))
:bypass t)


;;; Mission Leg Objects
(register-event ’mlo-set-information "MissionLegObject" "setInformation"
’(("String" "key")

   ("String" "value")))

(register-event ’create-mission-leg-object "MissionLegObject" "<init>"
())


;;; Mission Movement Objects
(register-event ’mmo-set-information "MissionMovementObject" "setInformation"
’(("String" "key")

   ("String" "value")))

(register-event ’create-mission-movement-object "MissionMovementObject" "<init>"

 ())

;;; Mission Sortie Objects
(register-event ’mso-set-information "MissionSortieObject"
"setInformation" ’(("String" "key") ("String" "value")))

(register-event ’create-mission-sortie-object "MissionSortieObject" "<init>"
())


;;; Mission Planner Listener
(register-event ’set-current-point "MissionPlannerListener" "setCurrentPoint"
’(("Point" "point")))


(register-event ’add-new-event-internal "MissionPlannerListener" "addNewEventInternal"
’(("Vector" "theVector")))


(register-event ’mpl-action-performed "MissionPlannerListener" "actionPerformed"
’(("ActionEvent" "actionEvent")))


;;; Mission Leg Editor
(register-event ’mle-action-performed "MissionLegEditor" "actionPerformed"
’(("ActionEvent" "actionEvent")))


;;; Mission Viewer Listener
(register-event ’mvl-action-performed "MissionViewerListener" "actionPerformed"
’(("ActionEvent" "actionEvent")))


;;; Mission Form
(register-event ’close-form "MissionForm" "closeForm"
’() :output-type "Vector")
```

```
;;; Mission Info Form
(register-event ’retrieve-leg "MissionInfoForm" "retrieveLegInfo"
() :output-type "MissionLegObject")
(register-event ’retrieve-movement "MissionInfoForm" "retrieveMvmtInfo"
() :output-type "MissionMovementObject")
(register-event ’retrieve-sortie "MissionInfoForm" "retrieveSortieInfo"
() :output-type "MissionSortieObject")
(register-event ’retrieve-info "MissionInfoForm" "retrieveInfo"
() :output-type "Hashtable")


;;; Create Mission Action
(register-event ’Create-Mission-Action-Action-Performed "CreateMissionAction" "actionPerformed"
’(("ActionEvent" "actionEvent"))
:bypass t)


;;; It’s not clear that we need this
;;; It’s the event when you click on the map
;;;   It’s followed by an ActionPerformed on Mission Planner Listener
;;;    that action is either PointChoiceCmd if you clicked on a recognized point
;;;                              or NewPointCmd if you clicked on something random
(register-event ’create-mission-event-point "CreateMissionAction" "createMissionEventPoint"
’(("Point" "point")))


;;; Edit Mission Action
(register-event ’edit-mission-action-action-performed "EditMissionAction" "actionPerformed"
’(("ActionEvent" "ActionEvent")))


;;; Publish Mission Action
(register-event ’publish-mission-action-action-performed "PublishMissionAction" "actionPerformed"
’(("ActionEvent" "theActionEvent")))


;;; Receive Mission Action
(register-event ’receive-mission-action-action-performed "ReceiveMissionAction" "actionPerformed"
’(("ActionEvent" "actionEvent")))


;;; Save Mission Action
(register-event ’save-mission-action-action-performed "SaveMissionAction" "actionPerformed"
’(("ActionEvent" "actionEvent")))

;;; Request Handler
(register-event ’request-handler-action-performed "RequestHandler" "actionPerformed"
’(("ActionEvent" "actionEvent")))


;;; The ife gui util Util stuff -used in load image


(register-event ’load-image "mil.af.afrl.ife.cmapi.util.Util" "loadImage"
’(("Component" "component")

   ("URL" "theUrl"))
:output-type "Image"
:static t

   :bypass "theImage")

(register-event ’load-image "mil.af.afrl.ife.gui.util.Util" "loadImage" ’(("Component"
"component")
   ("URL" "theUrl"))
:output-type "Image"
:static t

    :bypass "theImage")
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Method Accessors
;;; This defines the Lisp accessors to Java Class Data
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define-java-accessors Mission-Builder "MissionBuilder"
(get-information |getInformation| (("String" key)) :output-type "String")
 (get-leg-id |getLegID| (("String" key)) :output-type "String")
 (get-mission-event |getMissionEvent| (("String" key)) :output-type "MissionEventObject")
 (get-mission-events |getMissionEvents| () :output-type "Hashtable")
 (get-mission-leg |getMissionLeg| (("String" key)) :output-type "MissionLegObject")
 (update-mission-leg |updateMsnLeg| (("String" key) ("MissionLegObject" "theLeg")))
 (update-mission-sortie |updateMsnSrt| (("String" key) ("MissionSortieObject" "theSortie")))
 (update-mission-movement |updateMsnMvmt| (("String" key) ("MissionMovementObject" "theMovement")))
 (get-mission-movement |getMissionMovement| (("String" key)) :output-type "MissionMovementObject")
 (get-mission-sortie |getMissionSortie| (("String" key)) :output-type "MissionSortieObject")
 (get-msn-plot-objects |getMsnPlotObjects| () :output-type "Vector")
 (get-number-of-legs |getNumberOfLegs| () :output-type "int")
 (get-number-of-movements |getNumberOfMovements| () :output-type "int")
 (get-number-of-sorties |getNumberOfSorties| () :output-type "int")
 (get-number-of-refuel-events |getNumberOfRefuelEvents| () :output-type "int")
 (create-Mission-object |createMissionObject| () :output-type "MissionObject")
 (create-addtional-mission-info-panel |createNewAdditionalMissionInfoPanel|
             (("Vector" "currentInfoVector")))
   (set-client-panel |setClientPanel| (("ClientPanel" "clientPanel")))
   (get-event-ctr |getEventCtr| () :output-type "int")
   (set-event-ctr |setEventCtr| (("int" counter)))
   )


(defun get-number-of-events (mb)

(ht-size (get-mission-events mb)))

       (defun create-mission-builder (&key client-panel hash-table)
           (make-java-instance "MissionBuilder"

               (cond (client-panel ‘(("ClientPanel" ,client-panel)))
           (hash-table ‘(("Hashtable" ,hash-table))))))

(define-java-accessors MissionObject "MissionObject"

    (create-metadata |createMIOMetadata| () :output-type "String")

    (create-payload |createMIOPayload| () :output-type "String")

    (get-plot-objects |getMissionEvents| () :output-type "Vector")

     )

(define-java-accessors MissionLegObject "MissionLegObject"
   (leg-get-info |getInfo| () :output-type "Hashtable")
    (leg-get-Information |getInformation| (("String" key)) :output-type "String")
    (leg-set-information |setInformation| (("String" key) ("String" value))))

(defun create-mission-leg-object ()

    (make-java-instance "MissionLegObject"))

(define-java-accessors MissionMovementObject "MissionMovementObject"

 (movement-get-info |getInfo| () :output-type "Hashtable")

 (movement-get-Information |getInformation| (("String" key)) :output-type "String") (movement-set-
    information |setInformation| (("String" key) ("String" value))))
```

```
(defun create-mission-movement-object ()
 (make-java-instance "MissionMovementObject"))
(define-java-accessors MissionSortieObject "MissionSortieObject"
    (sortie-get-info |getInfo| () :output-type "Hashtable")
    (sortie-get-Information |getInformation| (("String" key)) :output-type "String")
     (sortie-set-information |setInformation| (("String" key) ("String" value))))
(defun create-mission-sortie-object ()
 (make-java-instance "MissionSortieObject"))
(define-java-accessors Mission-Event-Object "MissionEventObject"
    (event-get-info |getInfo| () :output-type "Hashtable")
    (event-get-Information |getInformation| (("String" key)) :output-type "String")
    (event-set-information |setInformation| (("String" key) ("String" value))))
(define-java-accessors Create-Mission-Action "CreateMissionAction"
 (get-Mission-Planner-Listener |getMissionPlannerListener| () :output-type "MissionPlannerListener")
 (set-current-mission-builder |setCurrentMissionBuilder| (("MissionBuilder" mb)))
(get-current-mission-builder |getCurrentMissionBuilder| () :output-type "MissionBuilder"))
(define-java-accessors Save-Mission-Action "SaveMissionAction"
   (create-xml |createXML| (("String" layer-name)) :output-type "String"))
(define-java-accessors Mission-Planner-Listener "MissionPlannerListener"
   (set-first-point |setFirstPoint| (("boolean" yes-or-no)))
    (set-current-point |setCurrentPoint| (("Point" point)))
    (add-new-event-internal |addNewEventInternal| (("Vector" v)))
    (get-event-counter |getEventCounter| () :output-type "int")
    (set-event-counter |setEventCounter| (("int" count))))
(define-java-accessors Pad-Editor "PadEditor"
  (get-pad |getPad| () :output-type "Pad"))
(define-java-accessors PAD "PAD"
   (add-draw-listener |addPadDrawObjectListener| (("PadDrawObjectListener" listener)))
  (remove-draw-listener |removePadDrawObjectListener| (("PadDrawObjectListener" listener))))
(define-java-accessors pad-action "PadAction"
  (want-draw-object-event? |wantDrawObjectEvent| ()))
(define-java-accessors cmapi "CMAPI"
   (get-layers |getLayers| () :output-type "Vector"))
(define-java-accessors cmapi-layer "mil.af.afrl.ife.cmapi.Layer"
   (layer-get-name |getName| () :output-type "String")
(get-plot-objects |getPlotObjects| () :output-type "Vector"))

(define-java-accessors Cmapi-Util "mil.af.afrl.ife.cmapi.util.Util"

 (load-image-1 |loadImage| (("Component" component) ("String" url))
    :output-type "Image" :static t)

      (load-image-2 |loadImage| (("Component" component) ("URL" url))
    :output-type "Image" :static t))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Client Frames and Panels
;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define-java-accessors Client-Frame "ClientFrame"

   (frame-get-client-panel |getClientPanel| () :output-type "ClientPanel"))


(define-java-accessors Client-Panel "ClientPanel"

      (set-mission-mode |setMissionMode| (("boolean" yes-or-no)))

      (get-mission-mode |getMissionMode| () :output-type "boolean")

      (get-actions |getActions| () :output-type "Hashtable") (get-cmapi |getCMAPI| () :output-type "CMAPI")

       (get-pad-editor |getPadEditor| () :output-type "PadEditor") )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Alternative Load Image Library
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(eval-when (:compile-toplevel :execute :load-toplevel)

    (register-class "PJAToolkit" "com.eteks.awt"))

(define-java-accessors eteks-awt "PJAToolkit"

   (create-image-string |createImage| (("String" url-string)) :output-type "Image")

    (create-image-url |createImage| (("URL" url)) :output-type "Image"))

(defun make-eteks-toolkit () (make-java-instance "PJAToolkit" ()))


;;; This is a java class of accessors to a variety of MAF system functions

 (eval-when (:compile-toplevel :execute :load-toplevel)

 (register-class "MafSupport" "edu.mit.aire.awdrat"))

(define-java-accessors MafSupport "MafSupport"
 (jbi-login |JBILogin| (("ClientFrame" client-frame)) :static t)
 (mdr-login |MDRLogin| (("ClientFrame" client-frame)) :static t)
 (do-action |doAction| (("ClientFrame" client-frame) ("String" action-string))
         :static t))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;
;;;; The system model per se
;;;
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(define-ensemble maf-editor
      :entry-events :auto
      :inputs () :outputs (the-model)
      :components ((startup :type maf-startup :models (normal compromised))
  (create-model :type maf-create-model :models (normal compromised))
  (create-events :type maf-create-events :models (normal compromised))
  (save :type maf-save :models (normal compromised)))

       :controlflows ((before maf-editor before startup)
      (after startup before create-model))

       :dataflows ((the-model create-model the-model create-events)
       (the-model create-events the-model save)
        (the-model save the-model maf-save-model))
               ;; The resources used, their modes, and their a priori likelihood of being in each mode
               :resources ((imagery image-file (normal .7) (hacked .3))
(code-files loadable-files (normal .8) (hacked .2)))
        ;; this maps which resources are used by which components of the computation
        :resource-mappings ((startup imagery)
(create-model code-files)
 (create-events code-files)
(save-model code-files))
        ;; this maps the conditional probabilities between the compromises and the misbehaviors
 :model-mappings ((startup normal imagery normal .99)
         (startup compromised imagery normal .01)
         (startup normal imagery hacked .9)
         (startup compromised imagery hacked .1)


         (create-model normal code-files normal .99)
         (create-model compromised code-files normal .01)
         (create-model normal code-files hacked .9)
         (create-model compromised code-files hacked .1)


         (create-events normal code-files normal .99)
         (create-events compromised code-files normal .01)
         (create-events normal code-files hacked .9)
         (create-events compromised code-files hacked .1)


         (save normal code-files normal .99)
         (save compromised code-files normal .001)
         (save normal code-files hacked .01)
         (save compromised code-files hacked .999))


       :vulnerabilities ((imagery reads-complex-imagery)
           (code-files loads-code)
           ))


(define-ensemble maf-startup
      :entry-events (startup)
       :exit-events (startup)
       :allowable-events (post-validate create-client-frame
         center-action load-image)
       :inputs ()
       :outputs ())


;;; Need to pick predicates for distinguishing these
 ;;; The compromise possible in startup is an image file
 ;;; attack leading to out of code execution?
 (defbehavior-model (maf-startup normal)
       :inputs ()
       :outputs ()
       :prerequisites ()
       :post-conditions ())


(defbehavior-model (maf-startup compromised)

 :inputs ()

 :outputs () :prerequisites ()

 :post-conditions ())

;;; Need defbehaviors for each of these even if its empty

(define-ensemble maf-create-model
       :entry-events (create-mission-action-action-performed)
        :exit-events (mission-builder-submit)
         :allowable-events (create-mission-builder-with-client-panel
             create-mission-builder
             create-mission-builder-with-hash-table
             mission-builder-submit
             (set-initial-info exit (the-model nil))
             create-mission-action-action-performed
             retrieve-info
             create-mission-action-action-performed
             (set-initial-info entry)
             )

       :inputs ()
       :outputs (the-model))

(defbehavior-model (maf-create-model normal)

       :inputs ()

        :outputs (the-model)

         :prerequisites ()

          :post-conditions ([dscs ?the-model mission-builder good])

          )

(defbehavior-model (maf-create-model compromised)

       :inputs ()

 :outputs (the-model)

 :prerequisites ()

 :post-conditions ([not [dscs ?the-model mission-builder good]])

 )

(define-ensemble maf-create-events
      :entry-events :auto
       :exit-events ()
       :allowable-events ()
       :inputs (the-model)
 :outputs (the-model)
:components ((get-next-cmd :type maf-get-next-cmd :models (normal))
  (get-event-info :type maf-get-event-info :models (normal compromised))
  (add-event-to-model :type maf-add-event-to-model :models (normal compromised))
  (get-leg :type maf-get-leg :models (normal compromised))
  (get-movement :type maf-get-movement :models (normal compromised))
  (get-sortie :type maf-get-sortie :models (normal compromised))
  (add-additional-info-to-model :type maf-add-additional-info :models (normal compromised))
  (continue :type maf-create-events :models (normal compromised)))

        :dataflows ((the-model maf-create-events the-model join-exit-exit)
         (the-model maf-create-events the-model add-event-to-model)
         (the-cmd get-next-cmd cmd more-events?)
         (the-event get-event-info the-event add-event-to-model)
         (the-model add-event-to-model the-model join-events-non-take-off)
         (the-event get-event-info event takeoff?)
         (the-leg get-leg the-leg add-additional-info-to-model)
        (lms-event-counter get-leg event-number add-additional-info-to-model)
         (the-movement get-movement the-movement add-additional-info-to-model)
         (the-sortie get-sortie the-sortie add-additional-info-to-model)
         (the-model add-event-to-model the-model add-additional-info-to-model)
         (the-model add-additional-info-to-model the-model join-events-take-off)
         (the-model join-events the-model continue)
         (the-model continue the-model join-exit-recur)
         (the-model join-exit the-model maf-create-events)
         )

            :controlflows ((after more-events?-build-event before add-event-to-model)
        (after more-events?-exit before join-exit-exit)
        (after takeoff?-get-additional-info before get-leg)
        (after takeoff?-get-additional-info before get-movement)
        (after takeoff?-get-additional-info before get-sortie)
        (after takeoff?-exit before join-events-non-take-off))

        :splits ((more-events? maf-more-events? (cmd) (build-event exit))
          (takeoff? maf-takeoff? (event) (get-additional-info exit)))
        :joins ((join-events (the-model) (take-off non-take-off))
        (join-exit (the-model) (recur exit)))

               ;; The resources used, their modes, and their a priori likelihood of being in each mode
                :resources ((code-files loadable-files (normal .8) (hacked .2)))
                ;; this maps which resources are used by which components of the computation
                :resource-mappings ((get-event-info code-files)
(add-event-to-model code-files)
(get-leg code-files)
(get-movement code-files)
(get-sortie code-files)
(add-additional-info-to-model code-files)
(continue code-files))

        ;; this maps the conditional probabilities between the compromises and the misbehaviors
        :model-mappings ((get-event-info normal code-files normal .99)
          (get-event-info compromised code-files normal .01)
          (get-event-info normal code-files hacked .9)



           (get-event-info compromised code-files hacked .1)

          (add-event-to-model normal code-files normal .99)
          (add-event-to-model compromised code-files normal .01)
          (add-event-to-model normal code-files hacked .9)
          (add-event-to-model compromised code-files hacked .1)


          (get-leg normal code-files normal .99)
          (get-leg compromised code-files normal .001)
          (get-leg normal code-files hacked .01)
          (get-leg compromised code-files hacked .999)
          (get-movement normal code-files normal .99)
          (get-movement compromised code-files normal .001)
          (get-movement normal code-files hacked .01)
          (get-movement compromised code-files hacked .999)


          (get-sortie normal code-files normal .99)
          (get-sortie compromised code-files normal .001)
          (get-sortie normal code-files hacked .01)
          (get-sortie compromised code-files hacked .999)


          (add-additional-info-to-model normal code-files normal .99)

           (add-additional-info-to-model compromised code-files normal .001)

           (add-additional-info-to-model normal code-files hacked .01)

           (add-additional-info-to-model compromised code-files hacked .999)

          (continue normal code-files normal .99)
          (continue compromised code-files normal .001)
          (continue normal code-files hacked .01)
          (continue compromised code-files hacked .999))

        :vulnerabilities ((code-files loads-code))
        )

(defbehavior-model (maf-create-events normal)

 :inputs (the-model)

 :outputs (the-model)

 :prerequisites ([dscs ?the-model mission-builder good])

 :post-conditions ([dscs ?the-model mission-builder good])

 )

(defbehavior-model (maf-create-events compromised)

         :inputs (the-model)

         :outputs (the-model)

         :prerequisites ([dscs ?the-model mission-builder good])

         :post-conditions ([not [dscs ?the-model mission-builder good]])

 )

(define-ensemble maf-get-next-cmd

      :entry-events (next-cmd)

      :exit-events ((next-cmd exit (the-cmd)))

      :inputs ()

      :outputs (the-cmd))

(defbehavior-model (maf-get-next-cmd normal)

      :inputs ()

      :outputs (the-cmd)

      :prerequisites ()
      :post-conditions ())

           (define-ensemble maf-get-event-info
                 :entry-events (create-mission-event-point)
                 :allowable-events (set-current-point
                     (create-mission-event-point exit)
                      create-mission-event-object
                      meo-set-information
                       mpl-action-performed
                        close-form
                       add-new-event-internal)
        :exit-events ((got-event-info exit (the-event)))
        :inputs ()
        :outputs (the-event))


(defbehavior-model (maf-get-event-info normal)

      :inputs ()

      :outputs (the-event)

      :prerequisites ()

      :post-conditions ([dscs ?the-event event good]))

(defbehavior-model (maf-get-event-info compromised)

      :inputs ()

      :outputs (the-event)

      :prerequisites ()

      :post-conditions ([not [dscs ?the-event event good]]))

(define-ensemble maf-add-event-to-model
      :entry-events (update-msn-evt)
      :allowable-events
      ((update-msn-evt exit (mb event-number event))
         add-new-event-internal
         create-new-additional-mission-info-panel
         )

        :exit-events (mpl-action-performed)
        :inputs (the-event the-model)
        :outputs (the-model event-number))


(defbehavior-model (maf-add-event-to-model normal)
      :inputs (the-event the-model)
      :outputs (the-model event-number)
      :prerequisites ([dscs ?the-event event good]
      [dscs ?the-model mission-builder good])
      :post-conditions
      ([add-to-map (events ?the-model)?event-number ?the-event
  ?before-maf-add-event-to-model]
         [dscs ?the-model mission-builder good]))


(defbehavior-model (maf-add-event-to-model compromised)

      :inputs (the-event the-model)

      :outputs (the-model event-number)

      :prerequisites ([not [dscs ?the-event event good]]

      [not [dscs ?the-model mission-builder good]])
```

```
    :post-conditions ([dscs ?the-model mission-builder good]))

(define-ensemble maf-get-leg

     :entry-events (retrieve-leg)

     :exit-events ((retrieve-leg exit (nil the-leg lms-event-counter)))

     :allowable-events (create-mission-leg-object mlo-set-information)

     :inputs ()

     :outputs (the-leg lms-event-counter))

(defbehavior-model (maf-get-leg normal)

     :inputs ()

     :outputs (the-leg lms-event-counter)

     :prerequisites ()

     :post-conditions ([dscs ?the-leg leg good]))

(defbehavior-model (maf-get-leg compromised)

     :inputs ()

     :outputs (the-leg lms-event-counter)

     :prerequisites ()

     :post-conditions ([not [dscs ?the-leg leg good]]))

(define-ensemble maf-get-movement
     :entry-events (retrieve-movement)
     :exit-events ((retrieve-movement exit (nil the-movement)))
     :allowable-events
      (create-mission-movement-object mmo-set-information)
      :inputs ()
      :outputs (the-movement))

(defbehavior-model (maf-get-movement normal)
      :inputs ()
      :outputs (the-movement)
      :prerequisites ()
      :post-conditions ([dscs ?the-movement movement good]))

(defbehavior-model (maf-get-movement compromised)
      :inputs ()
      :outputs (the-movement)
      :prerequisites ()
      :post-conditions ([not [dscs ?the-movement movement good]]))

(define-ensemble maf-get-sortie
      :entry-events (retrieve-sortie)
      :exit-events ((retrieve-sortie exit (nil the-sortie)))
      :allowable-events
      (create-mission-sortie-object mso-set-information)
      :inputs ()
      :outputs (the-sortie))

(defbehavior-model (maf-get-sortie normal)
      :inputs ()
      :outputs (the-sortie)
      :prerequisites ()
      :post-conditions ([dscs ?the-sortie sortie good]))

(defbehavior-model (maf-get-sortie compromised)
      :inputs ()
      :outputs (the-sortie)
```

```
        :prerequisites ()
        :post-conditions ([not [dscs ?the-sortie sortie good]]))

(define-ensemble maf-add-additional-info
        :entry-events ((retrieve-sortie exit))
        :exit-events (Mission-builder-add-info)
        :inputs (the-model the-leg the-movement the-sortie event-number)
        :outputs (the-model))

(defbehavior-model (maf-add-additional-info normal)
        :inputs (the-model the-leg the-movement the-sortie event-number)
        :outputs (the-model)
        :prerequisites ([dscs ?the-leg leg good]
          [dscs ?the-movement movement good]
          [dscs ?the-sortie sortie good]
          [dscs ?the-model mission-builder good])
                :post-conditions ([add-to-map (legs ?the-model) ?event-number ?the-leg
    ?before-maf-add-additional-info]
                    [add-to-map (sorties ?the-model) ?event-number ?the-sortie
    ?before-maf-add-additional-info]
                                          [add-to-map (movements ?the-model) ?event-number ?the-movement
    ?before-maf-add-additional-info]
                    [dscs ?the-model mission-builder good]))

(defbehavior-model (maf-add-additional-info compromised)
        :inputs (the-model the-leg the-movement the-sortie event-number)
        :outputs (the-model)
        :prerequisites ([dscs ?the-leg leg good]
          [dscs ?the-movement movement good]
          [dscs ?the-sortie sortie good]
          [dscs ?the-model mission-builder good])
                                              :post-conditions ([not [dscs ?the-model mission-builder good]]))

(defsplit maf-more-events? (cmd)
    (build-event (equal ?cmd 'new-event))
    (exit (equal ?cmd 'save-mission)))


(defsplit maf-takeoff? (event)

     (get-additional-info (take-off-event? ?event))

      (exit (not (take-off-event? ?event))))

(define-ensemble maf-save

      :inputs (the-model)

      :outputs ())

(defbehavior-model (maf-save normal)

      :inputs (the-model)

      :outputs ()

      :prerequisites ([dscs ?the-model mission-builder good])

      :post-conditions ([dscs ?the-model mission-builder good]))

(defbehavior-model (maf-save compromised)


 :inputs (the-model)


 :outputs ()


 :prerequisites ([dscs ?the-model mission-builder good])
 :post-conditions ([not [dscs ?the-model mission-builder good]]))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; attack models
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(define-attack-model maf-attacks

      :attack-types ((hacked-image-file-attack .3) (hacked-code-file-attack .5))

       :vulnerability-mapping ((reads-complex-imagery hacked-image-file-attack)

        (loads-code hacked-code-file-attack)))

;;; rules mapping conditional probabilities of vulnerability and attacks

(defrule bad-image-file-takeover (:forward)
   if [and [resource ?ensemble ?resource-name ?resource]
   [resource-type-of ?resource image-file]
   [resource-might-have-been-attacked ?resource hacked-image-file-attack]]
   then [and [attack-implies-compromised-mode hacked-image-file-attack ?resource hacked .9 ]
                            [attack-implies-compromised-mode hacked-image-file-attack ?resource normal .1 ]])

(defrule bad-image-file-takeover-2 (:forward)
   if [and [resource ?ensemble ?resource-name ?resource]
   [resource-type-of ?resource code-memory-image]
   [resource-might-have-been-attacked ?resource hacked-image-file-attack]]
    then [and [attack-implies-compromised-mode hacked-image-file-attack ?resource hacked .9 ]
                            [attack-implies-compromised-mode hacked-image-file-attack ?resource normal .1 ]])

(defrule hacked-code-file-takeover (:forward)
   if [and [resource ?ensemble ?resource-name ?resource]
   [resource-type-of ?resource loadable-files]
   [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
   then [and [attack-implies-compromised-mode hacked-code-file-attack ?resource hacked .9 ]
       [attack-implies-compromised-mode hacked-code-file-attack ?resource normal .1 ]])

(defrule hacked-code-file-takeover-2 (:forward)
   if [and [resource ?ensemble ?resource-name ?resource]
   [resource-type-of ?resource loadable-files]
   [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
   then [and [attack-implies-compromised-mode hacked-code-file-attack ?resource hacked .9 ]
       [attack-implies-compromised-mode hacked-code-file-attack ?resource normal .1 ]])

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;
;;;;; Hacked Code file attacks

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defrule bad-code-file-takeover (:forward)
    if [and [resource ?ensemble ?resource-name ?resource]
    [resource-type-of ?resource code-file]
    [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
    then [and [attack-implies-compromised-mode hacked-code-file-attack ?resource hacked .9 ]
                            [attack-implies-compromised-mode hacked-code-file-attack ?resource normal .1 ]])

(defrule bad-code-file-takeover-2 (:forward)
    if [and [resource ?ensemble ?resource-name ?resource]
    [resource-type-of ?resource code-memory-image]
    [resource-might-have-been-attacked ?resource hacked-code-file-attack]]
    then [and [attack-implies-compromised-mode hacked-code-file-attack ?resource hacked .9 ]
                            [attack-implies-compromised-mode hacked-code-file-attack ?resource normal .1 ]])
```